Enunciados de Sistemas Operativos

Gorka Guardiola Múzquiz

Enrique Soriano Salvador

06/09/22

Puedes conseguir la última versión de este documento en:

https://github.com/honecomp/honecomp.github.io/raw/main/ex/ejercicios_ssoo.pdf

Contenidos

Anagrams

Escriba un programa en C para linux imprima una línea por cada conjunto de anagramas extraído de los argumentos, que sólo contendrá caracteres ascii. Debe indicar al final de la línea entre corchetes las letras que no están en la misma posición en todos. Considere que como mucho hay 100 parámetros.

Por ejemplo:

```
$> ./anagrams hola peralo aloh pelaro tras tiki polare
   hola aloh [hola]
   peralo pelaro polare [erlo]
$>
```

Los argumentos que no tienen anagramas no deben imprimirse.

El programa consistirá de un único fichero: anagrams.c.

Catletter

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare catletter.git
$>  git clone catletter.git catletter
$> cd catletter
$> vi catletter.sh

#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf catletter.tgz catletter.git
```

Etiqueta la versión definitiva con el tag v1.0.0

Escribe un script de shell llamado catletter.sh que para todos los ficheros con nombre acabado en .txt del directorio que se pasa como único argumento del script, concatene el contenido de los ficheros en otros ficheros. Los ficheros cuyo nombre comience con un carácter x, deben concatenarse en el fichero x.output.

Se debe mantener un orden alfanumérico en base al nombre completo del fichero para realizar las concatenaciones.

Mayúscula y minúscula se deben considerar iguales a la hora de elegir el fichero de salida (el fichero de salida se llamará con el carácter en minúscula).

Si los ficheros de salida (.output) existen, se deben borrar antes de la generación de los nuevos ficheros de salida. El borrado debe ser silencioso (no se deben escribir mensajes de error por el borrado de dichos ficheros).

Por ejemplo, supongamos que el directorio contiene únicamente dos ficheros que empiezan con 'c', que son Carta.txt y contrato.txt. El contenido del fichero resultante c.output será el contenido de los ficheros Carta.txt y contrato.txt (en ese orden).

Un ejemplo de ejecución:

```
$> echo uno dos tres > cometa.txt
$> echo one two three > Camion.txt
$> echo hola hola > cohete.txt
$> echo deadbeef > Avion.txt
$> ./catletter.sh .
$> ls *.output
a.output  c.output
$> cat a.output
deadbeef
$> cat c.output
one two three
hola hola
uno dos tres
$>
```

Chext

Escribe un programa llamado chext.c en C para linux que reciba tres parámetros. Los dos primeros argumentos deben ser "extensiones" (i.e. la parte del nombre después del último punto). El tercer argumento debe ser la ruta de un directorio.

El programa debe recorrer recursivamente el directorio, renombrando todos los ficheros convencionales cuyos nombres tengan la "extensión" definida por el primer argumento y que tengan permiso de lectura para alguien (dueño, grupo o el resto). Los ficheros se deben renombrar para que tengan la "extensión" definida por el segundo argumento.

Para renombrar, se debe usar:

man 2 rename

Un ejemplo de ejecución sería (teniendo en cuenta que todos los ficheros de /tmp/d tienen permiso de lectura para el dueño):

```
$> ls /tmp/d
d2
uno.txt
dos.txt
tres.editado.doc
$> ls /tmp/d/d2
cuatro.txt
cinco.mp3
$> chext txt TEXT /tmp/d
$> ls /tmp/d
d2
uno.TEXT
dos.TEXT
tres.editado.doc
$> ls /tmp/d/d2
cuatro.TEXT
cinco.mp3
$>
```

Chkvar

Escriba un programa chkvar.c en C para Linux que reciba un número par de parámetros, obligatoriamente mayor o igual que dos. Cada par de parámetros será un nombre de variable de entorno y su contenido. Si el contenido coincide con el que hay en el entorno en esa variable de entorno saldrá con éxito. En caso contrario (la variable no existe o ese no es su contenido) saldrá con estado de error, mostrando un mensaje por la salida de error como el que se muestra a continuación:

```
$ echo $USER $HOME $LANG
paurea /home/paurea C
$ ./chkvar USER paurea && echo bien
bien
$ ./chkvar NOEXISTE paurea && echo bien
error: NOEXISTE != paurea
$ ./chkvar USER paurea HOME /home/paurea && echo bien
bien
$ ./chkvar USER paurea HOME /home/pepe LANG es LANG C && echo bien
error: HOME != /home/pepe, LANG != es
$  ./chkvar
usage: ch varname varcontent [varname varcontent] ...
$  ./chkvar USER
usage: ch varname varcontent [varname varcontent] ...
```

Implementar un script col.sh de shell en sh para Linux.

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare col.git
$>  git clone col.git col
$> cd col
$> vi col.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf col.tgz col.git
```

En el commit que entregues habrá dos versiones del script, etiquetadas con tags: 1.0.0 y 2.0.0.

La primera versión se desarrollará en la rama principal.

La segunda versión se desarrollará como una rama diferente creada tras acabar la primera. Cuando esté terminada, se hará merge en la rama principal y finalmente se etiquetará un commit de la rama principal.

1) Versión 1 (1.0.0)

El script col.sh mirará todos los ficheros que están directamente en el directorio actual.

Si los ficheros se llaman xxx_yyy.col (donde xxx e yyy es cualquier cosa y cualquier número de caracteres que no sean _ ni punto), partirá dichos ficheros en xxx.1 yyy.2. El primero contendrá la primera columna del fichero y el segundo la segunda. Si los nuevos ficheros existen, se sobreescribirán. Los ficheros que no cumplan el patrón se ignorarán.

Esta versión no recibirá parámetros (recibirlos será un error).

2) Version 2 (2.0.0)

La segunda versión del fichero, recibirá como parámetro único obligatorio un directorio de trabajo. Además, si no hay ficheros que partir, dirá "no files" por la salida de error y saldrá con error.

Ambas versiones deben comportarse de forma razonable (devolver errores cuando corresponda etc.).

Recuerda hacer push de las dos ramas.

Concat

Implemente en C para Linux un programa concat.c que concatene N ficheros en un fichero destino. Recibe al menos dos argumentos, fichero origen y fichero destino. Si hay más de dos, los primeros serán origen y el último será destino. El programa debe terminar avisando como corresponda si no se le dan los argumentos necesarios o hay algún error.

No se pueden ejecutar programas externos o una shell.

Por ejemplo:

```
$ echo hola > a
$ echo adios > b
$ echo pepe > c
$./concat a
usage: concat file1 [file2 ...] filedest
$ ./concat a b c d
$ cat d
hola
adios
pepe
$
```

Escriba un programa en C que implemente una biblioteca para proporcionar tres estructuras de datos para almacenar coordenadas. El tipo de dato se debe llamar Coor. Cada Coor tendrá dos componentes enteras (x, y) y además una etiqueta que será una cadena de texto con una longitud máxima de 255 caracteres.

La biblioteca debe ofrecer tres estructuras de datos para almacenar coordenadas de ese tipo:

- List, con operaciones públicas para:

  crear una lista vacía
  preguntar si una lista está vacía
  agregar una entidad al inicio de una lista
  agregar una entidad al final de una lista
  determinar el primer elemento (o la "cabeza") de una lista
  determinar el último elemento (o la "cola") de una lista
  preguntar si una coordenada está en la lista pasando sus dos componentes (x,y), y si es así, conseguir el puntero a la misma (si hay varias, la primera de la lista)
  preguntar si una coordenada está en la lista pasando su etiqueta, y si es así, conseguir el puntero a la misma (si hay varias, la primera de la lista)
  eliminar una coordenada de la lista pasando su puntero
  preguntar el número de elementos de la lista
  destruir la lista

- Stack, con las operaciones para crearla, destruirla y las típicas de una pila (push y pop).

- Queue, con las operaciones para crearla, destruirla y las típicas de una cola (enqueue y dequeue).

La lista tiene que estar implementada como una lista doblemente enlazada y se debe mantener un puntero al primer y último nodo.

Las coordenadas y las tres estructuras de datos tienen que estar implementadas en un fichero llamado list.c. Se debe proporcionar un fichero de cabeceras list.h para usar la biblioteca.

Se debe proporcionar un programa de prueba implementado en un fichero main.c. El programa de prueba debe:

Crear una lista con todas las coordenadas, con ambas coordenadas iguales (a, a), desde (0,0) hasta (10,10), imprimirla, eliminar las coordenadas con x par, imprimirlas de nuevo, y destruirlas.
Crear una pila de coordenadas, con ambas coordenadas iguales (a, a), desde (0,0) hasta (10,10) e imprimirlas en orden inverso (empezando con a=10).
Crear una cola de coordenadas, con ambas coordenadas iguales (a, a), desde (0,0) hasta (10,10) e imprimirlas en orden creciente (empezando con a=0).

Se debe entregar un único fichero comprimido llamado coor.tgz que contenga los tres ficheros, main.c, list.c y list.h.
Escriba un programa en C llamado copybytes.c que admita dos argumentos obligatorios con dos rutas: un fichero origen y un fichero destino. Opcionalmente se puede dar un tercer argumento con un número entero positivo. Si se pasan sólo dos argumentos, el programa debe copiar todo el fichero origen al fichero destino. Si se pasa un tercer argumento, sólo debe copiar los bytes indicados en el mismo.

Si el fichero destino existe, se debe truncar. En otro caso, se debe crear.

Si la ruta de origen es '-', se deberá leer de la entrada estándar. Si la ruta destino es '-', se deberá escribir en la salida estándar.

Se recomienda probar el programa con ficheros grandes (p. ej. varios MB).

Un ejemplo de ejecución:

```
$> echo hola adios > /tmp/a
$> ./copybytes /tmp/a /tmp/b 2
$> cat /tmp/b
ho$> ./copybytes /tmp/a /tmp/b
$> cat /tmp/b
hola adios
$> ./copybytes /tmp/a -
hola adios
$> ./copybytes - - < /tmp/a
hola adios
$>
```

Gitbasic

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.

Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare dirs.git
$>  git clone dirs.git dirs
$> cd dirs
$> vi dirs.sh


#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf dirs.tgz dirs.git
```

En el commit que entregues habrá dos versiones del script, etiquetadas con tags: v_1.0.0 y v_2.0.0.

La primera versión se desarrollará en la rama principal.

La segunda versión se desarrollará como una rama diferente creada tras acabar la primera. Cuando esté terminada, se hará merge en la rama principal y finalmente se etiquetará un commit de la rama principal.

1) Version 1 (v_1.0.0)
El script dirs.sh mirará todos los ficheros que están directamente en el directorio actual y sólo trabajará con los ficheros cuyo nombre acabe en .dir
No debe recibir ningún argumento (si lo recibe se considerará un error y actuará en consecuencia).

El fichero .dir contendrá en su interior el nombre de un directorio y una línea por cada fichero que se debe crear en ese directorio:

```
$> cat b.dir
dirb
fich1 10 USER
fich2 3 HOME
```

Lo que debe de hacer es para cada fichero, crear un directorio cuyo nombre es el contenido de la primera línea y
luego un fichero por cada línea describiendo un fichero (nombre, número de las líneas, variable de entorno para el contenido).
Los ficheros contendrán n líneas numeradas según dice la línea y con el contenido variable de entorno
que dice la tercera columna del descriptor. Las columnas estarán separadas por tabuladores.

Por ejemplo:

```
$> ls
b.dir  dirs.sh
$> ./dirs.sh
$> ls
b.dir  dirb  dirs.sh
$> ls dirb
fich1 fich2
$> echo $USER
```

```
pepe
$> cat dirb/fich1
1 pepe
2 pepe
3 pepe
4 pepe
5 pepe
6 pepe
7 pepe
8 pepe
9 pepe
10 pepe
$> cat dirb/fich2
1 /home/pepe
2 /home/pepe
3 /home/pepe
```

2) Version 2 (v_2.0.0)

La segunda versión del fichero, recibirá como parámetros un conjunto de ficheros. Irá a dónde se encuentren
dichos ficheros y hará lo mismo que la primera versión pero sólo con ese fichero. Si hay otros ficheros .dir en el
mismo directorio los ignorará.

```
$> echo $LANG
C
$> cat /tmp/a.dir
dosa
fich1 3 LANG
$> cat /home/pepe/x/c.dir
dosc
otro 3 LANG
b 5 USER
$> ./dirs.sh /tmp/a.dir /home/pepe/x/c.dir
$> ls /tmp/dosa
fich1
$> ls /home/pepe/x/dosc
otro
$> cat /home/pepe/x/otro
1 C
2 C
3 C
```

Elems

Escriba un programa en C para linux que admita como argumentos nombres de variables de entorno. El programa debe escribir por su salida, uno por línea, cada elemento que contienen las variables de entorno especificadas. Se considera que los elementos de una variable de entorno están separados por el carácter ':' (como en el caso de la variable de entorno PATH). Si alguna de las variables especificadas no existe, el programa debe terminar con fallo.

Por ejemplo:

```
$> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/
local/games:/usr/local/go/bin/:/opt/puppetlabs/bin/:./:/snap/bin
$> echo $XDG_DATA_DIRS
/usr/local/share:/usr/share:/var/lib/snapd/desktop
$> ./elems PATH XDG_DATA_DIRS
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/games
/usr/local/go/bin/
/opt/puppetlabs/bin/
./
/snap/bin
/usr/local/share
/usr/share
/var/lib/snapd/desktop
$> export hola=adios
$> ./elems hola PATATA
adios
ERROR: var PATATA does not exist
$> echo $?
1
$>
```

El programa consistirá de un único fichero: elems.c.

Execargs

Escriba un programa en C llamado execargs que ejecute todos los comandos que se le pasen como argumentos. El primer argumento debe ser el número de segundos que hay que esperar entre la finalización de la ejecución de uno y la ejecución del siguiente. El resto de argumentos que se le pasan al programa serán rutas de ejecutables, opcionalmente con argumentos para ese comando si la string contiene espacios. No se puede ejecutar una shell ni utilizar system(3).

Se considerará error si el programa no recibe al menos dos argumentos o el primero no es un entero.

En caso de error se debe avisar como corresponda y acabar la ejecución (no se deben ejecutar los programas restantes).

Para esperar un número de segundos, convertir de string a entero y partir una string en trozos usando un separador, se recomienda la lectura de:

man 3 sleep

man 3 atoi

man 3 strtol

man 3 strtok_r

Un ejemplo de ejecución es el siguiente:

```
$> ./execargs 2 '/bin/echo hola' '/bin/echo adios'
hola
adios
$> ./execargs
usage: execargs secs command [command ...]
$> ls
execargs
execargs.c
execargs.o
$> ./execargs 1 /bin/ls '/bin/echo ya esta'
execargs
execargs.c
execargs.o
ya esta
$> ./execargs 1 /xx/ls '/bin/echo ya esta'
error: /xx/ls : No such file or directory
```

Execparams

Escriba un programa en C execparams para GNU/Linux que ejecute los comandos que se le pasan como argumentos, indicando si han terminado con éxito o no. Se debe ejecutar un comando tras otro, indicando el comando que se ejecuta y su resultado exactamente como se indica en el ejemplo.

El programa debe salir con éxito si todos los comandos tienen éxito, en otro caso debe acabar con fallo. Si no se proporcionan comandos, debe indicar el uso del programa y salir con fallo. Se debe considerar que los comandos se encuentran en el directorio /bin.

```
$ ./execparams
usage: execparams cmd [cmd ...]
$ ls
execparams
execparams.c
file1
$ ./execparams ls 'echo hola'
### CMD: ls
execparams
execparams.c
file1
### STATUS: SUCCESS
### CMD: echo hola
hola
### STATUS: SUCCESS
$ echo $?
0
$ ./execparams 'ls /noexiste' 'echo adios'
### CMD: ls /noexiste
ls: cannot access '/noexiste': No such file or directory
### STATUS: FAILURE
### CMD: echo adios
adios
### STATUS: SUCCESS
$ echo $?
1
$
```

El programa en C debe ejecutar los programas con la llamada al sistema execv(2). Para tokenizar una cadena, se debe usar la función strtok_r(3).

Execha1

Escriba en C para Linux un programa llamado execsha1 cuyo propósito es crear un resumen hash SHA-1 del resultado de
la concatenación de todos los ficheros regulares ejecutables de un directorio.

El orden de la concatenación debe ser el orden en el que se encuentran en el directorio (no hay que ordenar los ficheros de ninguna forma). El programa sólo admite un argumento para indicar el directorio sobre el que se quiere trabajar. Si no se le pasa ningún argumento, debe actuar sobre el directorio de trabajo actual.

El resumen SHA-1 debe calcularse ejecutando el comando sha1sum (1), que debe ejecutarse sin ningún argumento. Este comando lee su entrada estándar hasta que se acaba, para luego calcular el resumen SHA-1 de los datos leídos y escribirlo por su salida estándar.

La salida del programa tiene que ser, simplemente, los 20 bytes expresados en hexadecimal del resumen SHA-1 generado. No se puede escribir nada más, ni antes ni después del SHA-1. En particular, los datos extra que escribe el comando sha1sum a su salida (espacios, guión...), no deben aparecer a la salida del programa.

No se permite la ejecución de un shell ni el uso de la llamada system().

A continuación se muestra un ejemplo:

```
$> execsha1 /tmp/d
354f16ad11753a48d17adbc93e1f4edd20d87a78
$>
```

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init fgreat
$>  cd fgreat
$> vi fgreat.sh

#hago el script y demás
$>  cd /work
$> tar -czvf fgreat.tgz fgreat
```

Fgreat

Escribe un script de shell llamado fgreat.sh. Va a haber tres versiones de este script. La primera no recibe parámetros y se corresponderá
con una rama en git. La segunda recibe un parámetro y se corresponderá con otra rama en git. Ambas deben estar etiquetadas con una
tag (v0.0.1 la primera v0.0.2 la segunda).

La primera mostrará un comando para renombrar el fichero más grande del directorio actual (no recursivo) añadiendo al final .old.

La segunda sólo debe mostrar el nombre del fichero más grande del directorio actual. Si se pasa su parámetro optativo -r sólo debe mostrar el nombre del fichero más grande del directorio actual y de sus subdirectorios (sin el -r no será recursivo).

Finalmente habrá una versión que será descendiente de un merge de estas dos, etiquetada v0.1.0 y que se comportará como la primera (imprimiendo el comando para renombrar), pero con el parámetro -r buscará el fichero de forma recursiva.

Si sucede cualquier error o los parámetros son incorrectos, los scripts deben escribir el error por la salida de error y salir con estado de error.

Un ejemplo de ejecución:

```
$> cd /work/fgreat
$> git checkout v0.0.1
$> cd /tmp
$> /work/fgreat/fgreat.sh
mv /tmp/xxx.tmp.txt /tmp/xxx.tmp.txt.old
$> cd /work/fgreat
$> git checkout v0.0.2
$> cd /tmp
$> /work/fgreat/fgreat.sh -r
/tmp/zzz/ggg.txt
$> cd /work/fgreat
$> git checkout v0.1.0
$> cd /tmp
$> /work/fgreat/fgreat.sh -r
mv /tmp/zzz/ggg.txt /tmp/xxx/ggg.txt.old
```

Fileinfo

Clona el respositorio Git:

    https://gitlab.etsit.urjc.es/esoriano/fileinfo

Completa el fichero fileinfo.sh para que, dado el path de un fichero, imprima por su salida la siguiente información:

    * Tamaño en KB

    * Número de inodo

    * Número de enlaces duros

    * Si algún proceso lo tiene abierto para leer de él

    * Si algún proceso lo tiene abierto para escribir en él

    * Punto de montaje de su sistema de ficheros

    * Tipo de sistema de ficheros (como lo muestra el comando mount)

    * Si el sistema de ficheros está montado en modo Read-Only

    * Espacio libre en su sistema de ficheros

Si el argumento pasado al script es un enlace simbólico, se debe proporcionar la información anterior para el fichero al que hace referencia.

Si el fichero no existe, se debe imprimir un error y acabar como proceda.

El formato de la salida se tiene que ajustar exactamente a la del siguiente ejemplo:

```
$> ./fileinfo.sh /home/pepe/mifichero.txt
SIZE: 42 KB
INODE: 34242
HARD LINKS: 3
OPEN FOR READING: Yes
OPEN FOR WRITING: No
MOUNT POINT: /home
FS TYPE: ext4
READ-ONLY: No
FREE SPACE IN FS: 7.5G
$> ln -s /home/pepe/mifichero.txt X
$> ./fileinfo.sh X
SIZE: 42 KB
INODE: 34242
HARD LINKS: 3
OPEN FOR READING: Yes
OPEN FOR WRITING: No
MOUNT POINT: /home
FS TYPE: ext4
READ-ONLY: No
FREE SPACE IN FS: 7.5G
$>
```

Puede hacer todos los commits necesarios en el repositorio Git, pero al menos

tiene que haber uno etiquetado con el tag release_v1.0.0, que será la versión final del script.
Se recomienda la lectura de las páginas de manual:

man 1 stat
man 1 realpath

Debe entregar un fichero llamado fileinfo.tgz con todo el directorio del repositorio Git.

Filterin

Escribe un programa en C llamado filterin.c que acepte una expresión regular como primer argumento, un comando como segundo argumento, y a continuación un número indeterminado de parámetros para dicho comando. El comando indicado podrá estar en los directorios /bin o /usr/bin (en ese orden de prioridad).

El programa tiene imprimir las líneas resultantes de la ejecución del comando que le ha pasado como argumento (con sus argumentos), que leerá de su entrada estándar. Las líneas impresas han de ser filtradas, de tal forma que únicamente se impriman las que se ajustan a la expresión regular indicada.

No se permite ejecutar un shell o usar system().

Un ejemplo de ejecución podría ser:

```
$> ls
a
b
$> cat a
hola
pepe
caracola
$> cat b
holahola
adios
lepe
$> cat a b | ./filterin 'pE$' tr e E
pEpE
lEpE
$> cat a b | tr e E | egrep 'pE$'
pEpE
lEpE
$>
```

Findbig

Escribe un script de sh llamado findbig.sh que admite dos argumentos, un tamaño en bytes y un directorio e imprime por orden de tamaño (los más grandes primero) todos los ficheros regulares contenidos en el directorio y sus subdirectorios que exceden o son iguales a ese tamaño y que contienen texto, imprimiendo nombre y tamaño en bytes, uno por línea.

Si el programa recibe un único directorio como argumento, hará exactamente lo mismo, pero sin comprobar ningún límite de tamaño.
Si no se reciben argumentos, o el número de argumentos es mayor de dos, se debe imprimir el error por donde corresponda y salir con un estatus de fallo.

Para ver si contienen texto, usa file (man 1 file).

```
$ findbig.sh 32 /tmp/g
762   /tmp/g/git/slides.tex
324   /tmp/g/dis/2017/traspas/2.tiempo/slides.tex
321   /tmp/g/comp/2018/traspas/9/slides.tex
40    /tmp/g/comp/2019/traspas/6/slides.tex
37    /tmp/g/comp/2019/traspas/9/slides.tex
36    /tmp/g/comp/2018/traspas/6/slides.tex
32    /tmp/g/sot/2018/shell/slides.tex
32    /tmp/g/dis/2018/traspas/2.tiempo/slides.tex
32    /tmp/g/comp/2019/traspas/0/slides.tex
32    /tmp/g/comp/2017/traspas/9/slides.tex
```

Crea tres repositorios trabajo, casa y central.git.
El repositorio central.git será de tipo bare y trabajo y casa serán clones de éste.

Los clones se realizarán a través de ssh.
Estos repositorios van a contener dos ficheros de texto con palabras para la prácticas de ripper, palabras1 y palabras2. Estos ficheros se añadirán en trabajo y casa indistintamente y se editarán en paralelo de manera que surjan conflictos y se resuelvan. La versión final de los tres repositorios debe estar sincronizada y contener al menos dos merge resultados de pull y push entre los repos y el central.

Con los repositorios en el mismo directorio (si hace falta cópialos ahí):

```
$ ls

trabajo casa central.git
$ tar -czvf gitbasic.tgz trabajo casa central.git
paurea@zeta00:~/a/a$ tar -czvf gitbasic.tgz trabajo casa central.git
trabajo/
casa/
central.git/
$
```

Es decir, usa:

```
tar -czvf gitbasic.tgz trabajo casa central.git
```

para generar el fichero de entrega, gitbasic.tgz

Greper0

Escriba un programa en C llamado greper.c que reciba dos o más argumentos. El primer argumento será el fichero de salida, y el resto de argumentos deben ser expresiones regulares. El programa debe escribir en el fichero de salida todas las líneas de los ficheros con extensión '.txt' del directorio de trabajo actual que encajan con alguna de las expresiones regulares. Para ello, tendrá que ejecutar el comando egrep.

El programa no debe escribir nada por su salida estándar ni ningún error escrito por egrep.

Si el fichero de salida no existe, se debe crear. Si existe, se debe truncar.

No se permite ejecutar un shell o usar system().

Un ejemplo de ejecución:

```
$> cat uno.txt
/home/pepe
/home/manuel
/home/pepin
$> cat dos.txt
hola
adios
$> ./greper salida 'm[Aa]n' 'ol'
$> cat salida
/home/manuel
hola
$>
```

Greper1

Escriba un programa en C llamado greper.c que recibe como argumentos pares formados por una regexp y un path. Por cada par, el programa debe escribir si el fichero indicado contiene o no contiene líneas con la regexp correspondiente o si ha ocurrido algún error a la hora de comprobarlo.

En caso de no recibir argumentos, el programa debe avisar pos la salida de error y fallar.

Se debe salir con estatus de éxito sólo en caso de que todos los ficheros contengan texto que encaje con su regexp correspondiente.

El programa debe ser todo lo concurrente posible (esto es, no debería ejecutar secuencialmente) y debe mostrar el resultado para todos los pares que ha recibido como argumentos por la salida estándar y un mensaje de error por la salida de error cuando se detecte.

Se recomienda la lectura de:

man 1 grep

El formato de la salida del programa se tiene que ajustar a este ejemplo:

```
$> cat /tmp/a
hola amigo
que tal
$> cat /tmp/b
no no no no
no no
no
$> cat /tmp/noexiste
cat: /tmp/noexiste: No such file or directory
$>./greper 'hola+' /tmp/a  'noest[abc]' /tmp/b  '^pepe' /tmp/noexiste
/tmp/a matches 'hola+'
/tmp/b does not match 'noest[abc]'
/tmp/noexiste: ERROR
$> ./greper
usage: greper regexp path [regexp path …]
```

Grepmatrix

Escribe un programa en C para GNU/Linux llamado grepmatrix que reciba como argumentos los siguientes datos:

* una lista de palabras a buscar en ficheros.

* una lista de rutas de ficheros en los que buscar las palabras.

La separación entre estas dos listas se delimitará con el modificador -f. Al menos debe darse una palabra y un fichero. La aparición de más de un modificador -f (o la de cualquier otro que no sea -f) resultará en un error.

La salida del programa debe indicar en una matriz qué palabras aparecen en cada fichero, siguiendo el formato exacto de los ejemplos que se pondrán a continuación. Se indicará con 'x' cuando aparece la palabra en el fichero, y con 'o' cuando no aparece. Observa que las columnas de la matriz están separadas por un único tabulador.

Para ver si un fichero contiene una palabra, el programa debe ejecutar fgrep(1). Se deben proporcionar los modificadores necesarios para que fgrep no escriba nada por su salida (ni datos ni errores).

Si hay error buscando alguna palabra en un fichero, el programa no debe escribir la matriz por su salida, debe escribir un error indicando qué fichero lo ha provocado
y terminar la ejecución, pero sin dejar ningún proceso huérfano.

Todas las búsquedas deben ser concurrentes.

Ejemplos:

```
$ ./grepmatrix
usage: grepmatrix word [words ...] -f file [file ...]
$ ./grepmatrix hola -f
usage: grepmatrix word [words ...] -f file [file ...]
$ ./grepmatrix  -f f1
usage: grepmatrix word [words ...] -f file [file ...]
$ echo hola adios > f1
$ echo adios > f2
$ ./grepmatrix  hola adios -f f1 f2
"hola"      "adios"
x     x     f1
o     x     f2
$ echo uno dos > f3
$ ./grepmatrix  uno dos tres hola adios -f f1 f2 f3
"uno" "dos" "tres"      "hola"      "adios"
o     o     o     x     x     f1
o     o     o     o     x     f2
x     x     o     o     o     f3
$ rm f2
$ ./grepmatrix  uno dos tres hola adios -f f1 f2 f3
error processing file f2
$
```

Hasword

Escriba un programa en C para Linux hasword.c que dado un conjunto de parejas de argumentos ejecute el comando fgrep para ver si el segundo argumento de la pareja (una palabra) está en el fichero cuyo nombre es el primer argumento de la pareja. Si el programa recibe un número impar de argumentos debe dar un error y acabar su ejecución. Si fgrep tiene un error procesando una pareja concreta tiene que escribir la cadena 'error'.

Para evitar que fgrep escriba cosas por su salida, se pueden usar sus opciones -q y -s.

Ejemplo:

```
$> hasword /tmp/ficheroa pepe ../ficherob patata /noexiste bla /tmp/existe oca
pepe: si
patata: no
bla: error
oca: si
$> hasword /tmp/patata
usage: hasword [[file word]...]
$>
```

Entregue un fichero hasword.c con el fuente del programa.

Ifaces

Escribe un programa llamado ifaces en C para Linux que escriba por su salida el número de interfaces de red que hay configuradadas en el sistema. No se permite ejecutar un shell ni usar funciones como system(3). Si hay algún problema ejecutando los programas externos o de otro tipo, el programa debe escribir una descripción del error como corresponda y salir con fallo.

Por ejemplo:

```
$> ifconfig | egrep '^[^ \t]' | wc -l
4
$> ifaces
4
$>
```

Escriba un programa includes.c en C para Linux que imprima por su salida todos los includes de los ficheros cuyo nombre termina en ".c" o ".h", omitiendo repetidos y en orden alfabético.

Se puede suponer que todos los includes tienen los mismos espacios.

Se prohibe ejecutar un shell o usar la función system.

```
$> ls
a.c
a.h
c.c
$> ./includes
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
$>
```

Jar

Implementar un script jar.sh de shell en sh para Linux.

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare jar.git
$>  git clone jar.git jar
$> cd jar
$> vi jar.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf jar.tgz jar.git
```

En el commit que entregues habrá dos versiones del script, etiquetadas con tags: 1.0.0 y 2.0.0.

La primera versión se desarrollará en la rama principal.

La segunda versión se desarrollará como una rama diferente creada tras acabar la primera. Cuando esté terminada, se hará merge en la rama principal y finalmente se etiquetará un commit de la rama principal.
Recuerda hacer push de las ramas y de las tags.

1) Versión 1 (1.0.0)

El script recibirá n argumentos (al menos 2 obligatoriamente). El primero será el classpath (directorios separados por dos puntos), el segundo un conjunto de ficheros en java.

El script extraerá los imports de todos los ficheros en java y se encargará de ver si hay ficheros .class que los resuelvan. Debe ignorar los imports que vienen de la librería estándar (los que empiezan por java).

Para cada import escribirá (uno por import y la lista final en orden alfabético) por la salida (el separador de columnas es un tabulador) el import, los ficheros de java separados por comas sin espacio y el fichero class completamente cualificado, es decir, un path absoluto (pueden ser también varios paths separados por comas si el import incluye un *)

```
a.b.c.Xxx   hola.java,mas.java      /home/paurea/src/java/a/b/c/Xxx.class
```

Si alguno de los imports no consigue resolverse, el script escribirá ?? en la tercera columna (el resto actuará bien) y al final saldrá con error y escribirá un error por la salida de error.

Por ejemplo:

```
$ pwd
/home/paurea/src/java
$ ./jar.sh /usr/lib:/home/paurea/lib:. *.java
com.paurea.List   hola.java,mas.java
      /home/paurea/src/java/com/paurea/List.class
com.paurea.Noexiste     hola.java,mas.java      ??
com.paurea.util.* hola.java
      /home/paurea/src/java/Runner.class,/home/paurea/src/java/com/paurea/util/
Killer.class
```

2) Version 2 (2.0.0)

Este script hará lo mismo que el anterior, pero tendrá en cuenta que puede haber ficheros .class metidos dentro de
ficheros .jar, la última columna puede ser un jar en lugar de un class.
Ojo, en este caso, el classpath puede contener ficheros jar directamente o dentro del directorio especificado
(javac -cp algs4.jar:. *.java).

Para mostrar el contenido de los ficheros .jar puedes usar el comando:
jar tfv algs4.jar

Por ejemplo:
$ pwd
/home/paurea/src/java
$ ./jar.sh /home/paurea/lib:. *.java
com.paurea.List   hola.java,mas.java
      /home/paurea/src/java/com/paurea/List.class
com.paurea.util.* hola.java
      /home/paurea/src/java/Runner.class,/home/paurea/src/java/com/paurea/util/Killer.class
edu.princeton.cs.algs4.StdOut ejercicio.java   /home/paurea/lib/algs4.jar

Se pueden suponer que sólo se importarán clases e ignorar los genéricos.

Killof

Escribe un script de shell killof.sh que reciba como parámetro una lista de directorios (si no recibe
ninguno, se supone que es el directorio actual). El script debe buscar todos los procesos
pertenecientes al usuario (el que ejecuta el script) que tienen ficheros abiertos en esos directorios (y sus subdirectorios).
Para cada uno de esos procesos, debe escribir el comando para matarlos (sin ejecutarlo).

Por ejemplo:

```
$> mkdir /tmp/e /tmp/o
$> touch /tmp/e/a /tmp/o/b
$> tail -f /tmp/e/a &
[1] 3437
$> tail -f /tmp/o/b &
[2] 3439
$> killof.sh /tmp/e /tmp/o
kill -KILL 3437
kill -KILL 3439
$>
```

Se harán las siguientes excepciones:

    Si el nombre del fichero comienza con un punto, se ignorará ese fichero. P.ej.: /home/pepe/x/.mifichero se debería ignorar, pero /home/pepe/.config/blah no se debería ignorar

    Si el nombre del fichero solo contiene letras mayúsculas, se le mandará la señal SIGINT en lugar de SIGKILL
El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init killusers
$>  cd killusers
$> vi killusers.sh

#hago el script y demás
$>  cd /work
$> tar -czvf killusers.tgz killusers
```

Killusers

Escribe un script de shell llamado killusers.sh que reciba un conjunto de directorios como parámetro.
El script matará todos los procesos que tengan el directorio y subdirectorios como directorios de trabajo.
Adicionalmente, puede recibir un primer parámetro optativo (-d) para que en lugar de matar los procesos, escriba por su salida estándar el comando que ejecutaría. Si no hay ningún proceso que tenga esos directorios como directorio de trabajo, el programa no hará nada. Si alguno de los directorios que se le pasan no existe, debe salir con error y escribir "error: dir XXXX not found", donde XXX es el nombre del directorio por su salida de error y parar en ese parámetro. Ojo, hay que ser cuidadoso con el directorio actual en el script. El programa no debe suicidarse, ni matar ninguno de sus comandos (ni escribir comandos que lo hagan).

Etiqueta el script definitivo con la tag v1.0.0. No se pide ninguna estructura en el git más allá de esta etiqueta en el commit que se considere la entrega.
Escriba un programa en C para Linux lines.c que admita como argumentos una string y un directorio. Para todos los ficheros convencionales del directorio terminados en ".txt", debe buscar todas las líneas que contengan la string usando el comando externo fgrep. Todas esas líneas tienen que escribirse en el fichero lines.out en el directorio padre del que se ha pasado como argumento. El fichero se debe crear, y si existe, se debe truncar y sobrescribir.

El comando fgrep no debe escribir ningún error, aunque los haya. En caso de error, tu programa solo debe escribir en la salida de errores una línea describiendo el error y salir con el status de fallo, borrando antes el fichero de salida lines.out. Lee la página de manual de fgrep para ver qué status de salida usa cuando tiene errores (no cuando no encuentra líneas que se ajusten al patrón).

Ej:

```
$> mkdir z
$> ls -l / > z/ls.txt
$> echo usr es el mejor dir > z/a.txt
$> echo nada de nada > z/b.txt
$> ./lines usr z
$> cat lines.out
usr es el mejor dir
drwxr-xr-x  14 root     root        4096 feb 23  2018 usr
$>
```

Listdir

Escriba un programa en C para Linux listdir.c que dado un único argumento que especifica una ruta de un directorio, liste todas sus entradas menos las que comienzan por punto. Las entradas del directorio se escribirán incluyendo la ruta pasada como argumento.

No se permite ejecutar ningún programa externo.

SOLO SE DEBE ENTREGAR EL FICHERO FUENTE.

Por ejemplo:

```
$> ls -a  /etc/samba
.
..
gdbcommands
smb.conf
tls
$> listdir /etc/samba
/etc/samba/gdbcommands
/etc/samba/smb.conf
/etc/samba/tls
$>
```

Listpath

Escribe un programa en C para GNU/Linux llamado listpath.c que, para todos los directorios especificados en la variable de entorno PATH, indique cuántos ficheros
convencionales contiene siguiendo el mismo formato que el ejemplo.

Si en algún momento el programa tiene algún error, se debe abortar la ejecución avisando del error como corresponda.

Ejemplo:

```
$> echo $PATH
/usr/local/go/bin:/home/esoriano/bin:/home/esoriano/.local/bin:/usr/local/
sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/
snap/bin:/usr/local/plan9port/bin:/home/esoriano/bin
$> ./listpath
/usr/local/go/bin: 2
/home/esoriano/bin: 139
/home/esoriano/.local/bin: 3
/usr/local/sbin: 0
/usr/local/bin: 8
/usr/sbin: 302
/usr/bin: 2643
/sbin: 143
/bin: 149
/usr/games: 7
/usr/local/games: 0
/snap/bin: 0
/usr/local/plan9port/bin: 257
/home/esoriano/bin: 139
$>
```

Rúbrica (sobre 10)

La calificación dependerá de los aspectos estudiados durante el curso que el alumno ya debería conocer:

Si no compila o tiene warnings con las flags del curso (shadow, all) no se considera entregado.

Funcionamiento: si no se imprime el número de ficheros covencionales que tiene cada directorio, la nota máxima de la prueba será de 6.0 puntos. En caso de no funcionar correctamente o incumplir la especificación del resto del enunciado (teniendo en cuenta las dos excepciones especificadas anteriormente), la nota será menor o igual de 3.0 puntos.

- Liberación de recursos: +1 puntos
- Manejo de errores: +2 puntos
- Uso correcto de las llamadas a sistema y bibliotecas: +2 puntos.
- Código: lenguaje C, estilo, legibilidad, tabulación, uso de funciones, factorización, nivel de anidación, etc., +2 puntos, -2 puntos.

El trabajo es individual y debe ser original.

ListTS

Convierte tu implementación de la lista (List) de coordenadas del ejercicio 1 (list.c y list.h) para que sea thread safe. La biblioteca no debe incluir los tipos Stack y Queue del ejercicio 1, sólo debe implementar List.

Se debe usar la primitiva mutex de la librería pthreads. Múltiples hilos de pthreads deben ser capaces de usar una misma lista de forma concurrente. Cada lista creada debe usar su propio mutex para asegurar su acceso concurrente (esto es, no se debe usar un mutex global para todas las listas que se creen usando la biblioteca).

Se debe proporcionar un programa de prueba implementado en un fichero main.c. El programa de prueba debe:

Crear una lista vacia.
Crear 100 threads. A cada thread se le debe pasar como argumento una string, que usará como etiqueta en sus coordenadas.
Cada uno de los threads (1) insertará en la lista 100 coordenadas (las que quiera, pero todas con la etiqueta correspondiente); (2) después de insertar las coordenadas, deberá encontrar 50 coordenadas de las suyas (con su etiqueta) y eliminarlas de la lista.
El hilo principal esperará a que terminen todos los hilos. Después comprobará que el número de elementos de la lista es correcto (100*50), destruirá la lista y terminará.

Se debe entregar un único fichero comprimido llamado coorthreads.tgz que contenga los tres ficheros, main.c, list.c y list.h.

Logpairs

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare logpairs.git
$>  git clone logpairs.git logpairs
$> cd logpairs
$> vi logpairs.sh

#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf logpairs.tgz logpairs.git
```

Escribe dos scripts de shell. El primero guardará una lista de la pareja de usuarios que ejecutan más procesos a la vez (se apuntará una pareja cada vez que ejecute).
El segundo mostrará con qué usuario ha aparecido otro en dicha lista y cuándo.

El primer script logpairs.sh recibe como único argumento optativo un fichero que contiene
una lista de usuarios a ignorar, uno por línea (si no hay fichero, no se ignora ningún usuario).

El segundo, withuser.sh, recibe un nombre de usuario obligatorio y procesa la salida del primer script.

El primer script añade una línea cada vez que ejecuta al fichero userpairs.txt que se encuentra en el directorio de trabajo. Si el fichero no existe, lo crea y si ya existe, le añade la línea. Esa línea contiene los dos usuarios que más procesos están ejecutando en ese momento (sin contar los que están en el fichero de usuarios a ignorar) y la fecha de estar en un formato similar al que se ve a continuación:

```
$> cat userpairs.txt
paurea miller 20_Jan_2019_20:12
paurea miller 20_Jan_2019_21:12
john ron 18_Feb_2019_22:17
john paurea 15_Aug_2019_01:00
```

Cada línea se corresponde con una ejecución de userpairs.txt.

El segundo script withuser.sh, recibe un usuario como argumento obligatorio e imprime por su salida estándar todos los usuarios que aparecen como pareja con él antes o después en la misma línea en el fichero userpairs.txt (que se busca en el directorio actual) y las fechas correspondientes, una detrás de otra.

Por ejemplo, con el script anterior:

```
$> withuser paurea
miller 20_Jan_2019_21:12 20_Jan_2019_20:12
john 15_Aug_2019_01:00
```

Ambos scripts se deben desarrollar en ramas separadas (uno puede estar en master). Las versiones definitivas deben estar etiquetadas con v.1.0.0.log y v.1.0.0.with respectivamente.

Escribe un script de sh llamado logusers.sh que admite un único argumento, que es el nombre de un directorio. Para cada usuario que esté trabajando en el sistema, debe crear un fichero llamado nombre-de-usuario.log en dicho directorio que contenga únicamente los PIDs de los procesos de dicho usuario, uno por línea.

Si no existe el directorio, se debe crear. Si existe, se debe considerar un error.

Si no se pasa un único argumento, o si sucede cualquier otro error, se debe imprimir el error por donde corresponda y salir con un estatus de fallo.
- markup.sh

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.

Entregarás un .tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare markup.git
$>  git clone markup.git markup
$> cd markup
$> vi markup.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf markup.tgz markup.git
```

Markup

Implementa un script markup.sh de shell en sh para Linux que recibe una cadena de texto (CADENA) y conjunto de ficheros (al menos uno) con extensión .mup.
Si no recibe la cadena y al menos un fichero o el fichero no acaba en .mup, se considerará un error.

Los ficheros contendrán texto demarcado por líneas que comiencen por
--CADENA TXT, --CADENA COMMENT (el formato es dos guiones, a continuación. sin espacio, la cadena,
que no contendrá espacios y luego, separado por un espacio una palabra,
TXT o COMMENT).
Si las marcas no se encuentran a principio de línea, se considerarán texto normal.
Un ejemplo de fichero pedro como cadena, podría ser:

```
Hola soy un fichero
--pedro TXThola estoy dentro
--pedro TXTmás texto dentro
texto fuera
--pedro COMMENThola estoy dentro
--pedro COMMENTacaba el comentario
otro comentario
--pedro COMMENThola estoy dentro
--pedro COMMENTacaba el comentario
--pedro TXTotro pedro
aqui acaba el fichero
```

Puede haber regiones con diferentes cadenas. Si la cadena no es la que está considerando el fichero, se considerará como texto normal (ver el ejemplo de juan abajo).

El script debe extraer los trozos delimitados por las marcas y en el caso de los comentarios (COMMENT) añadiendo # al principio y al final de las líneas correspondientes.

Puedes usar ficheros intermedios pero debe de ser posible ejecutar varias veces el script en paralelo sin que una instancia pise a otra.

En caso de error, el script debe imprimir su uso por la salida de error y terminar con error.

En caso de no encontrar marcas, el fichero de salida se creará vacío y no se considerará un error.

Por ejemplo (el $> es el prompt):

```
$> cat fich.mup
Hola soy un fichero
--pedro TXThola estoy dentro
--pedro TXTmás texto dentro
texto fuera--pedro TXT
--pedro COMMENThola estoy dentro comentario
--pedro COMMENTacaba el comentario --pedro TXT
otro comentario
--pedro COMMENT2hola estoy # dentro comentario
--pedro COMMENTacaba el comentario 2
--juan TXTotro pedro--pedro TXT
```

```
aqui acaba el fichero
$> cat otro.mup
--pedro COMMENThola
--pedro COMMENTsoy otro
$> ./markup pedro fich.mup otro.mup
$> echo $?
0
$> cat fich.txt
hola estoy dentro
más texto dentro
#hola estoy dentro comentario#
#acaba el comentario  --pedro TXT#
#2hola estoy # dentro comentario#
#acaba el comentario 2#
$> cat otro.txt
#hola#
#soy otro#
$> ./markup.sh > /dev/null
usage: ./markup.sh string fich.mup [fich.mup ..]
$> echo $?
1
$>
```

El commit con la versión definitiva tiene que tener un tag v1.0.0

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare medstats.git
$>  git clone medstats.git medstats
$> cd medstats
$> vi medstats.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf medstats.tgz medstats.git
```

El repositorio (bare) que entregues tendrá una tag v1.0.0 con la versión definitiva del script.
Ve haciendo commits mientras lo desarrollas. No es necesario que haya varias ramas.

El script recibe un conjunto de ficheros como parámetro. Los ficheros contendrán varias listas de números en columnas separadas por tabuladores en coma flotante y una línea de cabecera con palabras separadas por tabuladores.

En caso de no recibir parámetros o que alguno de ellos no se corresponda con un fichero, el script escribirá un mensaje por su salida de error y saldrá con error.

La cabecera comienza por el caracter #.
Por ejemplo:

```
$> cat fichero
#input      output      time
1.2   3.4   4.5
1.2   3.4   6.5
2.2   6.4   4.5
1.2   3.4   6.5
2.134 6.67  4.98
```

El script escribirá una línea para cada fichero de entrada con la mediana (el

valor central de los elementos ordenados o la media de los dos centrales si son pares) y la media aritmética (la suma dividida entre el número de elementos) de cada una de las columnas en el formato que muestra el ejemplo a continuación:

```
$> cat fichero1
#input      output      timeT
1.2   3.4   4.5
1.2   3.4   6.5
2.2   6.4   4.5
1.2   3.4   6.5
2.134 6.67  4.98
...
$> cat fichero2
#inputX     outputX     timeX
1.2   3.4   4.5
3.2   3.7   6.5
2.2   6.4   4.8
1.2   3.4   5.5
2.134 6.67  4.98
...
$> medstats fichero1 fichero2
fichero1:
      input:      avg:  12.00,      median:     45.65
      output:     avg:  12.00,      median:     12.68
      timeT:      avg:  12.00,      median:     13.67
fichero2:
      inputX:     avg:  12.23,      median:     17.60
      outputX:    avg:  34.43,      median:     45.80
      timeX:      avg:  12.00,      median:     13.67
```

En la salida del script (como en los ficheros) todos los caracteres invisibles son un sólo tabulador y los números en coma flotante tienen exactamente dos decimales.
Escribe un programa en C para Linux que funcione de forma similar al comando du(1), pero con menos opciones. El programa recibe los siguientes argumentos:

- El fichero de origen. Si es -, se leerá de la entrada estándar.

- El fichero destino. Si es -, se escribirá en la sa
Myenv: Variables de entorno

Myenv

Escribe un programa en C para GNU/Linux, myenv.c que reciba como argumentos nombres de variables de entorno.
El programa debe escribir por su salida el nombre de las variables junto con su valor.
Si una variable no está definida, debe darse un error y seguir procesando los argumentos.
En el caso de que haya al menos una variable no definida, el programa saldrá con error. En caso
contrario, saldrá con éxito.
Además, antes de escribir eso, debe escribir el UID y el PID del proceso.

Por ejemplo:

```
$> export BBB=hola
$> ./myenv HOME SHELL PATH AAA BBB
      UID: 43
      PID: 1243
      HOME: /home/al-31-32/pepe
      SHELL: /bin/bash
      PATH: /bin/:/root/bin
      error: AAA does not exist
      BBB: hola
$>
```

Los errores deben escribirse por la salida de error.
La entrega consistirá únicamente en un fichero myenv.cEjercicio 5: Myfgrep, ficheros

Escriba un programa en C llamado myfgrep.c que admita un mínimo de dos argumentos. De los N argumentos recibidos, los primeros N-1 serán palabras a buscar en un fichero. El último argumento es una ruta a un fichero.

El programa escribirá, una única vez, las líneas que contengan en cualquier posición alguna de las palabras como subcadena en el formato del ejemplo (número de línea, dos puntos, y la línea).

En caso de encontrar alguna línea que imprimir saldrá con éxito. En caso de no encontrar ninguna saldrá con estado 1. En caso de haber cualquier error (permisos, argumentos...) saldrá con estado 2 y escribirá un mensaje de error por la salida estándar de error.

Un ejemplo de ejecución:

```
$> echo 'hola adios
bla ble bli
xholap adios' > /tmp/a
$> myfgrep hola zz /tmp/a
1:hola adios
3:xholap adios
$> myfgrep hola adios /tmp/a
1:hola adios
3:xholap adios
$> echo $?
0
$> myfgrep rr zz /tmp/a
$> echo $?
1
$> myfgrep rr
```

```
usage: myfgrep word [word]... file
$> echo $?
2
```

Mysplit

Escriba un programa mysplit.c en C para Linux similar al comando split, que reciba como argumentos obligatorios un tamaño N en bytes y un fichero. El programa cortará el fichero en varios ficheros que contegan como mucho N bytes y cuyos nombres que sean el nombre del fichero original precedido por un número de 3 dígitos que comienza en 000 y representa el orden en el que aparecen los datos de ese fichero en el original.
Si alguno de los ficheros que se va a crear existen, se truncarán y sobreescribirán (no será un error).
Si no hubiese suficientes argumentos o fuesen incorrectos (no existe el fichero, etc.), escribirá
su uso por la salida de error y saldrá con estado erróneo.

Por ejemplo:

```
$> seq 0 9 | tr -d '\n' > fich
$> mysplit 3 fich
$> ls -l
total 20
-rw-rw-r-- 1 paurea paurea  3 nov 11 10:55 000fich
-rw-rw-r-- 1 paurea paurea  3 nov 11 10:55 001fich
-rw-rw-r-- 1 paurea paurea  3 nov 11 10:55 002fich
-rw-rw-r-- 1 paurea paurea  1 nov 11 10:55 003fich
-rw-rw-r-- 1 paurea paurea 10 nov 11 10:55 fich
$> cat 000fich
012$> cat 001fich
345$> cat 002fich
678$> cat 003fich
9$>


$> mysplit 7
usage: mysplit N file
```

Nota: asegúrate de probarlo con ficheros grandes, por ejemplo:

```
$> cp /usr/bin/ls .
$> split 30000 ls
$> cat 0*ls > newls
$> cmp ls newls   #no dice nada, mi split funciona en este caso
$>
```

Tenemos las notas de unos alumnos en unos ficheros
llamados ejercicio[1-4].txt. Esos ficheros contienen dos columnas
separadas por tabuladores o espacios: un nombre de usuario y una nota
(número entero).

Implemente un script exam.sh que a partir de esos cuatro ficheros
imprima la lista de usuarios (ordenada alfabéticamente) con todas sus
notas separadas por tabuladores y al final su nota máxima. La salida debe
respetar el formato del ejemplo (cabecera, mayúsculas y tabuladores).

Si un usuario no está en alguno de los ficheros, se entenderá que su
nota para ese ejercicio es un 0.

Por ejemplo:

```
:$ cat ejercicio1.txt
pepe 1
```

```
pepa    3
:$ cat ejercicio2.txt
pepe 10
pepa 4
:$ cat ejecicio3.txt
juan 5
pepa 9
:$ cat ejercicio4.txt
pepa 3
pepe 3
:$ ./exam.sh
NOMBRE    E1    E2    E3    E4    MAXIMA
juan    0    0    5    0    5
pepa    3    4    9    3    9
pepe    1    10    0    3    10
:$
```

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init notas
$>  cd notas
$> vi notas.sh

#hago el script y demás
$>  cd /work
$> tar -czvf notas.tgz notas
```

Notas

Escribe un script de shell llamado notas.sh que reciba un conjunto de ficheros como parámetro. En cada fichero
habrá dos columnas, la primera será el el dni (con letra) y las notas. El script creará un fichero NOTAS con la nota final (la media de todas las notas con un decimal) y NP si el alumno no ha presentado alguna de las notas (no está presente en alguno de los ficheros).

Adicionalmente, el script dará un error (y saldrá con el consiguiente status) si cualquiera de las notas no está entre 0.0 y 10.0.

Por ejemplo:

```
$> cat notas1
234234W 10.0
346456F 7.5
$> cat nota2
234234W 1.0
$> notas.sh notas1 notas2
$> cat NOTAS
#dni nota
234234W 5.5
346456F NP
```

Dict

Implementa en C para GNU/Linux una librería y un programa main
para probar un diccionario. Las cadenas se buscarán utilizando como
clave de búsqueda otra cadena y se almacenarán en una tabla dict
con desbordamiento (es decir, encadenamiento separado, con una lista
simplemente enlazada para almacenar las claves que colisionan).

La función dict que utilizará la implementación será esta función (función has de djb):

```
unsigned int
dict(unsigned char *str)
{
      unsigned int dict = 5381;
      char *ps;

      for (ps = str; *ps != '\0'; ps++)
      {
            dict = (dict << 5) + dict + *ps;
      }
      return dict;
}
```

Nótese que el rango de la función hash es diferente al de la tabla.
Tienes que elegir un tamaño de tabla y definir con una constante,
por ejemplo, 1024. Usa la operación módulo para ajustar el tamaño.

La librería tendrá dos operaciones:

```
int insert(NSdict *dict, char *key, char *value);
```

Registra un valor, value, asociado a una clave key en el diccionario dict,
devolviendo -1 si la clave ya existe o hay un error, y cero en otro caso.

```
char *lookup(NSdict *dict, char *key);
```

Devuelve el valor asociado a la clave o NULL si la clave no existe.

El registro asociado no se borra de la tabla en ningún caso.

La operación:
`void freedict(NSdict *dict);`
libera el diccionario.

Escribe un programa main en un fichero separado (main.c) de prueba que reciba N argumentos. Cada pareja de argumentos se interpretará como clave valor. el último argumento será la clave para una búsqueda y se debe imprimir el valor o 'no presente' por la salida estándar en caso de que no esté. El programa saldrá con éxito en este caso.

Si la clave está repetida, se escribirá 'repetida' por la salida estándar y el programa saldrá con error. No se debe imprimir nada por la salida de error en ese caso. Si el número de argumentos es impar, se debe escribir un error por la salida de error y salir con error.

La entrega consistirá en un fichero .tgz con tres ficheros:
nsdict.c nsdict.h y main.c

Ejemplo de ejecución:

```
$ nsdict uno pepe dos juan tres pedro dos
juan
$ nsdict uno pepe dos juan tres pedro bla
no presente
$ nsdict uno pepe dos juan tres pedro juan
no presente
```

Offcat

Escriba un programa en C para linux que reciba dos parámetros obligatorios que sea un número y un fichero y escriba el contenido del fichero por la salida desde el offset descrito por el número.

Por ejemplo:
```
$> ./cat fichero
123456789
$> ./offcat 5 fichero
6789
$>
```

El programa también debe funcionar con ficheros binarios y ficheros de gran tamaño.

El programa debe escribir errores y salir con el status correspondiente si no recibe los parámetros adecuados, o sucede cualquier error.

El programa consistirá de un único fichero: offcat.c.

Photosren

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init photosren
$>  cd photosren
$> vi photosren.sh

#hago el script y demás
$>  cd /work
$> tar -czvf photosren.tgz photosren
```

Escribe un script de shell llamado photosren.sh que reciba un directorio como parámetro. Tiene que buscar todos los ficheros que son imágenes y meterlos en un subdirectorio cuyo nombre será la fecha actual en un formato similar a 20_jan_2020. Los nombres de los ficheros se renombrarán para que su nombre sea un número seguido de la extensión y todos los nombres sean de la misma longitud (sin incluir la extensión).

Para detectar qué ficheros son imágenes, se recomienda el uso del comando file:

```
man 1 file
```

El script debe recibir al menos un parámetro.

Por ejemplo:

```
$> date
lun mar 23 10:05:22 CET 2020
$> ls /tmp/x
a.gif
b.jpeg
c.txt
de.png
r.png
qwe.png
ss.gif
etc...
r
$> photosren /tmp/x
$> ls /tmp/x
23_mar_2020
c.txt
r
$> ls /tmp/x/23_mar_2020
000.gif
001.jpeg
002.png
003.png
004.png
005.gif
etc...
103.gif
$>
```

PingerC2

Escriba un programa en C llamado pinger.c que recibe como argumento un parámetro numérico, N, obligatorio y una serie de nombres de máquinas (al menos uno).

Tiene que averiguar si las máquinas están en línea utilizando el programa ping para mandarle dos ECHO_REQUEST. Si al menos una de las máquinas pasadas como argumento no está en línea (no contesta o tarda más de N segundos en contestar a 2 mensajes consecutivos), se escribirá su nombre por la salida de error y se saldrá inmediatamente (aunque queden otras máquinas que no han contestado todavía). El program debe ser lo más concurrente posible y debería ejecutar en menos de, aproximadamente, 2*N+1 segundos independientemente de sus argumentos.

El programa sólo debe escribir un error en caso de que sus argumentos sean erróneos (menos que los obligatorios, no hay nombres de máquinas) y si detecta alguna máquina offline. El programa debe salir con estado de error si ha encontrado una máquina que no contesta a ping o si sus argumentos son erróneos.

Ejecute ping con -q para que no imprima salida por cada mensaje (aunque ping seguirá escribiendo la cabecera y el resumen), más adelante veremos redirecciones para resolver este problema completamente.

Un ejemplo de ejecución:

```
$> pinger 2 www.google.com urjc.es
PING www.google.com (216.58.201.164) 56(84) bytes of data.

PING urjc.es (212.128.240.50) 56(84) bytes of data.

--- www.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 5.759/6.435/7.112/0.676 ms

--- urjc.es ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 3.527/3.863/4.199/0.336 ms

$> echo $?
0
$> pinger 3  www.google.com nannnnnanan.noeefefd.com bla.nosdsdfs.yaya

PING www.google.com (216.58.201.164) 56(84) bytes of data.
ping: nannnnnnannn.noeefefd.com: Name or service not known

no existe: nannnnnanan.noeefefd.com
$> echo $?
1
$>
```

PingerC

Escribe un programa en C para Linux que acepte como argumentos una serie de direcciones IP o nombres de DNS. Por cada dirección o nombre, debe comprobar si responde a ICMP haciendo un ping de un único mensaje y con timeout de 5 segundos. Para ello, deberá ejecutar el programa /bin/ping con los parámetros adecuados.

Todos los pings se deben ejecutar en paralelo. El programa debe acabar con éxito solo si todos los pings han funcionado correctamente.

Por ejemplo, suponiendo que esas direcciones no responden a ping:

```
$> date
mar oct  9 10:20:19 CEST 2018
$> ./pinger 199.21.3.4  199.21.3.5  199.21.3.6
PING 199.21.3.5 (199.21.3.5) 56(84) bytes of data.

--- 199.21.3.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
PING 199.21.3.6 (199.21.3.6) 56(84) bytes of data.

--- 199.21.3.6 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
PING 199.21.3.4 (199.21.3.4) 56(84) bytes of data.

--- 199.21.3.4 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$> echo $?
1
$> date
mar oct  9 10:20:25 CEST 2018
```

El programa consistirá de un único fichero: pinger.c.

PingerSH

Escribe un script de shell llamado pinger.sh. El script recibe un parámetro numérico obligatorio y una serie de ficheros (al menos uno).

Tiene que averiguar si las máquinas mencionadas en los ficheros están en línea. Si al menos una de las máquinas contenidas en el fichero no está en línea, se renombrará el fichero con su nombre acabado en .down. Se considerará que una máquina esta fuera de línea si no contesta a ping. Ping debe mandar el número de mensajes que describe el primer argumento pasado al script.

El script sólo debe escribir un error en caso de que sus argumentos sean erróneos (menos que los obligatorios, uno de los ficheros no existe) y si detecta alguna máquina offline. El script debe salir con estado de error si ha encontrado una máquina que no contesta a ping o si sus argumentos son erróneos.

Un ejemplo de ejecución:

```
$> cat funo
www.google.com
www.urjc.es
$> cat fdos
nannnnnanan.noeefefd.com
www.yahoo.com
$> pinger 2 funo fdos
no existe: nannnnnanan.noeefefd.com
$> echo $?
1
$>
```

Escribe un programa llamado pipeline.c en C para linux que reciba tres parámetros. Cada parámetro tiene que ser una string con un comando y sus argumentos. El programa debe esperar a todos los procesos y acabar con el estado de salida del último comando del pipeline.

Es necesario usar la llamada execv para ejecutar. Para tokenizar una string, se recomienda el uso de:

```
man 3 strtok_r
```

Los ejecutables de los comandos se deben buscar en los directorios /bin y /usr/bin (en ese orden).

El estado de salida debe ser el del último comando del pipeline.

Por ejemplo:

```
$> ls -l / | grep u | wc -l
4
$> pipeline 'ls -l /' 'grep u' 'wc -l'
4
$> pipeline 'echo hola' cat 'tr a-z A-Z'
HOLA
$>
```

Printvars

Escriba un programa en C llamado printvars que imprima por su salida el valor de la las variables de entorno que se le han pasado como argumentos (tendrá un número indeterminado de argumentos).

Si la variable de entorno no existe como se ha pasado, hay que probar (en este orden) si existe con todos sus caracteres en mayúsculas y también con todos en minúsculas. Si no se encuentra ninguna variable que encaje, se avisará que no existe la variable y se seguirán procesando argumentos, pero en este caso se debe informar al usuario del error y el comando no puede acabar con éxito.

Si no se pasan argumentos, se debe describir como se usa el programa y terminar con fallo.

La salida del programa debe seguir el mismo formato del ejemplo:

```
$> export HOLA=uuuu
$> export adios=aaaa
$> ./printvars
usage: printvars name [name ...]
$> ./printvars HOLA noexiste Adios
HOLA: uuuu
error, la variable 'noexiste' no existe
adios: aaaa
$> echo $?
1
$> ./printvars hola
HOLA: uuuu
$> echo $?
0
$>
```

Procinfo

Escriba un programa llamado procinfo.c en C para GNU/Linux que muestre los procesos que están ejecutando en el sistema. Para cada proceso, tiene que escribir por su salida la siguiente información:

* PID del proceso.
* Descriptores que tiene abiertos.
* Directorio de trabajo actual.
* Ruta del fichero que ejecuta el proceso.
* Resumen SHA1 del fichero que ejecuta el proceso.

El programa puede recibir cero o más argumentos. Si no se da ningún argumento, debe mostrar la información de todos los procesos del sistema. Si recibe argumentos, deben ser PIDs. En ese caso, se mostrará únicamente la información de los procesos indicados.

Si entre los argumentos hay alguno que no pueda ser un PID, el programa debe fallar indicando su forma de uso (sin mostrar información sobre ningún fichero) y salir con estatus de fallo. Si encuentra algún error procesando algún PID, debe continuar con el resto, pero no podrá acabar con estatus de éxito. Sólo se puede terminar con estatus de éxito si no se ha encontrado ningún error.

Para conseguir toda la información requerida, el programa debe inspeccionar los ficheros y directorios de /proc. Para calcular el resumen SHA1 se debe ejecutar el comando sha1sum(1). Para conseguir la ruta a la que apunta un enlace simbólico, se recomienda la lectura de la página de manual de la llamada readlink(2).

Se debe escribir la información exactamente en el mismo formato que los ejemplos de más abajo. Si hay algún tipo de error obteniendo la información de un proceso (por ejemplo por permisos), se debe escribir "ERROR" en su salida estándar en lugar de la información de ese proceso.

El programa debe escribir todos los errores que sean necesarios por su salida de errores.

Por ejemplo:

```
$ procinfo 123 125 421 2> /dev/null
123:
      ERROR
125:
      Descriptors:
            0: /tmp/x
            1: /tmp/y
            2: /dev/pts/2
      Current dir: /tmp
      Exec: /bin/cat
      SHA1: f6f536b25d7ee7e314944e41d885d9075f79a9a8
421:
      Descriptors:
            0: /dev/pts/0
            1: /dev/pts/1
            2: /dev/pts/2
      Current dir: /home/pepe
      Exec: /bin/ls
      SHA1: d3a21675a8f19518d8b8f3cef0f6a21de1da6cc7
$ procinfo 124 pepe 231
```

```
usage: procinfo [pid ...]
$
```

Se deben usar tabuladores para formatear la salida.

Se puede considerar que un proceso tiene un máximo de 256 descriptores de fichero.

Psuser

Escriba un programa en C para Linux llamado psuser.c que liste todos los procesos que está ejecutando el usuario que ejecuta el programa en este momento, similar a ejecutar (puede ejecutar los programas externos desde C):

```
ps aux | grep "^$USER "
```

Por ejemplo:

```
$  ps aux | grep "^$USER"
esoriano   19440  0.0  0.0  10700   532 pts/1    S+   11:28   0:00 sleep 1
esoriano   19442  0.0  0.0  14116  3312 pts/4    R+   11:28   0:00 ps aux
esoriano   19443  0.0  0.0  11660  2736 pts/4    S+   11:28   0:00 grep --
color=auto ^esoriano
$ ./psuser
esoriano   19440  0.0  0.0  10700   532 pts/1    S+   11:28   0:00 sleep 1
esoriano   19442  0.0  0.0  14116  3312 pts/4    R+   11:28   0:00 ps aux
esoriano   19443  0.0  0.0  11660  2736 pts/4    S+   11:28   0:00 grep --
color=auto ^esoriano
$
```

Recol

Hay que implementar un script recol.sh de shell en sh para Linux.

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare recol.git
$>  git clone recol.git recol
$> cd recol
$> vi recol.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf recol.tgz recol.git
```

En el commit que entregues habrá dos versiones del script, etiquetadas con tags: 1.0.0 y 2.0.0.

La primera versión se desarrollará en la rama principal.

La segunda versión se desarrollará como una rama diferente creada tras acabar la primera. Cuando esté terminada, se hará merge en la rama principal y finalmente se etiquetará un commit de la rama principal.

1) Versión 1 (1.0.0)

El script recol.sh mirará todos los ficheros que están directamente en el directorio actual y sólo trabajará con los ficheros cuyo nombre acabe en .data
Cada fichero contendrá datos de varios usuarios, con una cabecera indicando qué datos son y una columna para cada dato. Todos los separadores son tabuladores:

```
$> cat fich.data
user  hours_swimming    hours_cycling
paurea      3.4   2.1
alf   8.00  7.198
robx  2.4   .3
jeff  0.4   0.3
$> cat fich2.data
user  liters_coffe      liters_tea liters_water
paurea      2.1   .1    4.0
alf   3.1   5.1   2.1
robx  2.2   .3    6.7
jeff  .1    3.1   5.0
```

El script imprimirá por la salida estándar, para el usuario que ejecuta el script lo siguiente:

El nombre de los ficheros en los que está presente
Un resumen de los datos de el usuario en dichos ficheros. Todo esto en el formato del siguiente ejemplo:

```
$> whoami
paurea
$> recol
user is paurea
files: fich.data fich2.data
hours_swimming    hours_cycling     liters_coffe      liters_tea liters_water
3.4   2.1   2.1   .1    4.0
```

```
$>
```

Los datos provienen de los ficheros con sus nombres en orden alfabético (por eso el agua está a la derecha de las horas nadando) y fich.data aparece a la izquierda de fich2.data.

Si no está en ningún fichero, imprimirá por la salida de error no data y acabará con error:

```
$> recol
no data
$>
```

Esta versión no recibirá parámetros (recibirlos será un error).

2) Version 2 (2.0.0)

La segunda versión del fichero, recibirá como parámetro único obligatorio un nombre de usuario e imprimirá los datos
de ese usuario. Si en lugar de un nombre de usuario, recibe el modificador '-u' utilizará el usuario actual.

```
$> whoami
paurea
$> recol alf
user is alf
files: fich.data fich2.data
hours_swimming    hours_cycling     liters_coffe      liters_tea liters_water
8.00  7.198 3.1   5.1   2.1
$> recol -u
user is paurea
files: fich.data fich2.data
hours_swimming    hours_cycling     liters_coffe      liters_tea liters_water
3.4   2.1   2.1   .1    4.0
$>
```

Ambas versiones deben comportarse de forma razonable (devolver errores cuando corresponda etc.).

Recuerda hacer push de las dos ramas, tags, etc.

Renex

Escriba tres versiones de un script para renombrar ficheros. Las tres se llamarán renext.sh y estarán en un repositorio de Git.

La primera versión recibe dos argumentos obligatorios, el nombre de la extensión origen y destino. Todos los ficheros cuyo nombre acaba en un punto seguido de la extensión origen deberán renombrarse a la extensión destino en el directorio actual. Si el fichero destino existe, se sobrescribirá sin dar un error.

Si el fichero existe

```
$>   du -a
4    ./b.jpeg
0    ./z/b.jpeg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
8    ./z
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
$> renext.sh jpeg jpg
$>   du -a
0    ./z/b.jpeg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
8    ./z
4    ./b.jpg
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
```

La segunda versión hace lo mismo que la anterior, pero recibe un modificador -r adicional que hace que el comando actúe de forma recursiva, es decir, cambie los ficheros en todos los subdirectorios del árbol que cuelga del directorio actual. El modificador -r es opcional, sin él, se comportará como la versión anterior.

```
$>   du -a
4    ./b.jpeg
0    ./z/b.jpeg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
8    ./z
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
$> renext.sh -r jpeg jpg
$>   du -a
0    ./z/b.jpg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
```

```
8    ./z
4    ./b.jpg
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
```

La tercera versión recibe un parámetro obligatorio adicional (además del -r opcional) que es una expresión regular.
Sólo debe actuar sobre los ficheros cuyo nombre antes del último punto (por ejemplo para bla.txt sólo bla) encaje
con la expresión regular.

```
$>   du -a
4    ./b.jpeg
0    ./z/bla.jpeg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
8    ./z
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
$> renext.sh -r jpeg jpg 'b.*a'
$>   du -a
4    ./b.jpeg
0    ./z/bla.jpg
0    ./z/a.jpg
4    ./z/b.txt
4    ./z/a.txt
8    ./z
0    ./a.jpg
4    ./b.txt
4    ./a.txt
20    .
```

Se entregará un fichero tgz que contenga un repositorio git en el que la primera versión esté etiquetada con el tag v0.0.0, la segunda v0.1.0 y la tercera v1.0.0. El repositorio se llamará practica3. Desde el directorio padre,
se ejecutará

```
tar -czvf practica3.tgz practica3
```

para generar el fichero de entrega, practica3.tgz En los commits del repositorio (debería haber al menos 3, pero puede haber más) sólo debería estar el fichero renext.sh y un README del que no se hará commit (estará en el gitignore).

Renro

Implementa un script renro.sh de shell en sh para Linux.

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare renro.git
$>  git clone renro.git renro
$> cd renro
$> vi renro.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf renro.tgz renro.git
```

En el commit que entregues habrá dos versiones del script, etiquetadas con tags: 1.0.0 y 2.0.0.

Sólo habrá una rama.

La segunda versión se desarrollará en la misma rama. Cuando esté terminada, se etiquetará un commit de la rama principal.

1) Versión 1 (1.0.0)

El script renro.sh mirará todos los ficheros que están directamente en el directoripasado como parámetro.

Esta versión del script recibe un único directorio obligatorio. El script renombrará todos los ficheros que sean de sólo lectura (para el usuario, el grupo y todos) al nombre del fichero seguido de .ro (por ejemplo, el fichero xxx.gz se renombrará a xxx.gz.ro).

```
 $ ls -l zz
total 0
-r--rw-r-- 1 paurea paurea 0 abr 19 12:20 a.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.c.ll
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.txt
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 b.txt.yy
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 c.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:21 c.txt.oo
 $ ./renro.sh zz
 $ ls -l zz
total 0
-r--rw-r-- 1 paurea paurea 0 abr 19 12:20 a.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.c.ll
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.txt
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 b.txt.yy.ro
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 c.txt.ro
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:21 c.txt.oo
```

2) Version 2 (2.0.0)

Esta versión recibe dos parámetros obligatorios, el directorio de trabajo y una extensión. Si el nombre del fichero acaba en esa extensión (la parte después del

último punto, coincide), el fichero se ignorará.

```
 $ ls -l zz
total 0
-r--rw-r-- 1 paurea paurea 0 abr 19 12:20 a.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.c.ll
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.txt
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 b.txt.yy
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 c.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:21 c.txt.oo
 $ ./renro.sh zz txt
 $ ls -l zz
total 0
-r--rw-r-- 1 paurea paurea 0 abr 19 12:20 a.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.c.ll
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:20 b.txt
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 b.txt.yy.ro
-r--r--r-- 1 paurea paurea 0 abr 19 12:20 c.txt
-rw-rw-r-- 1 paurea paurea 0 abr 19 12:21 c.txt.oo
```

Recuerda hacer push de los tags

Repn

Escriba un programa repn.c en C para linux que reciba como argumento un número obligatorio y un conjunto de palabras (que puede ser
vacío).

Debe imprimir las primeras 5 letras de cada palabra (o la palabra entera si su longitud es menor de 5) repetida n veces (donde n es
el primer argumento). Cada argumento se escribirá en una línea separada. El programa recibirá, de forma opcional, (es decir,
puede no recibir este argumento, pero es obligatoria su implementación) como primer argumento antes del número la flag -r. En caso de recibirla, imprimirá las líneas al revés.

Por ejemplo:

```
$> repn 3 hola alpes apetito lata estudio
holaholahola
alpesalpesalpes
apetiapetiapeti
latalatalata
estudestudestud
$> repn -r 3 hola alpes apetito lata estudio

alohalohaloh
seplaseplasepla
itepaitepaitepa
atalatalatal
dutsedutsedutse
$> repn
usage: repn [-r] n [string ...]
$> repn 38
$>
```

Renexec

Escribe un script de shell llamado renexec.sh. El script recibe al menos un parámetro, que es el directorio en el que tiene que realizar el trabajo. Opcionalmente, puede recibir un segundo argumento con una extensión.

El script debe renombrar cada fichero que sea ejecutable (para cualquiera, grupo o dueño) con la extensión dada. Si ya tiene extensión, hay que quitar la vieja. Si no se ha pasado una extensión, debe renombrarlo con la extensión .exec y debe actuar en todo el subárbol bajo el directorio que se pase como parámetro.

Un ejemplo de ejecución:

```
$> ls -l /tmp/x/y
-rw------- 1 esoriano esoriano    0 feb 24 09:39  a
-rwxr----- 1 esoriano esoriano    0 feb 24 09:39  b.sh
-rw-r-xr-x 1 esoriano esoriano    0 feb 24 09:39  c
$> ./renexec.sh  /tmp/x executable
$> ls -l /tmp/x/y
-rw------- 1 esoriano esoriano    0 feb 24 09:39  a
-rwxr----- 1 esoriano esoriano    0 feb 24 09:39  b.executable
-rw-r-xr-x 1 esoriano esoriano    0 feb 24 09:39  c.executable
$>
```

Rerun

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare rerun.git
$>  git clone rerun.git rerun
$> cd rerun
$> vi rerun.sh
#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf rerun.tgz rerun.git
```

Escribe un script de shell que se ocupe de reejecutar ciertos programas cada vez que mueran. Los programas se le pasan como argumentos y sus nombres deben terminar en "_comando". El script debe aceptar un número de argumentos indeterminado.

Se puede suponer que los comandos van a tener un nombre único y no va a haber otro comando igual en el sistema. El script hará polling cada segundo y si alguno de los N comandos ha muerto lo volverá ejecutar. Se puede suponer que no hay dos instancias del mismo script ejecutando sobre los mismos comandos a la vez.

Si alguno de los comandos ya estaba ejecutando al comenzar el script, debe imprimir un error, salir con estado de fallo y no hacer nada.

A la hora de inspeccionar los procesos que están ejecutando, es importante tener en cuenta que algunos modificadores del comando ps muestran la ruta del fichero que ejecuta el proceso (si es un progama interpretado, muestra la ruta del del intérprete) y sus argumentos, mientras que otros modificadores de ps muestran únicamente el nombre del programa que ejecuta el proceso.

En caso de no poder ejecutar un programa porque no exista su ruta o no sea ejecutable (ya sea la primera vez, ya sea alguna de las subsiguientes), el script debe crear un fichero vacío con el nombre del comando acabado en ".fail" en el directorio de trabajo actual, escribir un mensaje de error "fail" y salir con estado de fallo.

El script debe recibir al menos un programa y como ya se ha dicho anteriormente, todos deben acabar en "_comando". En caso de no recibir los argumentos correctos, el script debe imprimir un error con su forma de uso y salir con estado de fallo.

Este es un posible ejemplo de ejecución:

```
$> mkdir pruebas
$> cd pruebas
$> echo '#!/bin/sh
sleep 100
' > primer_comando
$> cp primer_comando segundo_comando
$> chmod u+x *comando
$> /home/paurea/rerun.sh ./primer_comando ./segundo_comando &
$> ps|grep primer_comando
 9401 pts/0 00:00:00 primer_comando
$> kill 9401
$> ps|grep primer_comando
```

```
 9409 pts/0    00:00:00 primer_comando
$> rm primer_comando
$> kill 9409
$> ps|grep primer_comando
...espero
fail
$> ls
primer_comando.fail segundo_comando
$>
```

El script se debe desarrollar en una única rama. La versión final a entregar estará etiquetada mediante una tag v1.0.0rerun.

El fichero de entrega se debe llamar rerun.tgz

Ripper

Escriba un script de shell llamado "ripper.sh" encargado de realizar un ataque de diccionario a un conjunto de contraseñas.

El script recibirá dos argumentos obligatorios con el nombre de dos archivos. El primero contendrá una lista de contraseñas de usuario resumidas con el algoritmo SHA1. El formato del archivo será el siguiente:

```
login1:b9f310db36c930cf10f8d990972eb5971e4ee533
login2:4436a76d1c8a63c84bd6ec953e2e13c8563efcf3
```

El segundo archivo contendrá una lista de palabras en castellano (una por línea) que deben usarse para para realizar el ataque de diccionario.

Si no se proporcionan los argumentos necesarios o no existen los ficheros, el programa debe acabar con estatus de fallo y avisar por su salida de errores.

Para obtener el resumen SHA1 a partir de texto plano, se puede emplear el comando sha1sum(1). Si ejecutamos sha1sum sin argumentos, leerá los datos por su entrada estándar hasta el momento en que reciba EOF, y después escribirá el resumen SHA1 por salida estándar. Esto significa que hay que ejecutar sha1sum para cada resumen que se quiera calcular. Recuerde que un resumen SHA1 son siempre 40 caracteres que representan cifras hexadecimales.

Salida del programa: el script que se debe realizar mostrará por salida estándar los login y las contraseñas en claro que haya conseguido romper, siguiendo un formato similar al archivo de contraseñas:

```
$ ./ripper.sh  passwds dict
login2:guitarra
login1:piruleta
$
```

Nota: Recuerde que las contraseñas no deben incluir el carácter \n (salto de línea) al final de la línea. Dicho carácter debe ser eliminado. Como ejemplo puede considerar las palabras ’ejemplo’ y ’ejemplo\n’, cuyos respectivos SHA1 son:

```
a6c05bd5d579650bdb5f2088a3b8ea44452929a7
83389eabdee6084c080e9bb398c362bfd5d623bf
```

Por ejemplo:

```
$ cat words
uno
dos
tres
hola
adios
$ cat passwords
pepe:81b6f50734d17c2cfc160cfc07ec31b9bcd2a91e
juan:6e16b1cfc9620a2660786cb9a23227fdf03fc39d
manolo:cdc4e9f90112a90a27d8a6d267cfc5391bae3c6b
$ ./ripper.sh passwords words
pepe:uno
manolo:adios
$ ./ripper.sh
```

```
usage: ripper.sh passwordfile dict
$ ./ripper.sh /noexiste words
error: /noexiste does not exist
$ ./ripper.sh passwords
usage: ripper.sh passwordfile dict
$
```

Runcmds

Escriba un programa runcmds.c en C para GNU/Linux que lea líneas que contengan comandos (uno por línea) y los ejecute, filtrando su salida con una expresión regular. El programa tiene un primer parámetro, que es la expresión regular, y puede admitir segundo argumento que indique el fichero del que tiene que leer las líneas con los comandos. Si no se pasa el segundo argumento, debe leer líneas de comandos de su entrada estándar.

Si se leen las líneas de un fichero de comandos pasado como argumento, el programa se debe asegurar de que no se ejecute este programa sobre el mismo fichero de comandos hasta que haya terminado (esto es, no se pueden ejecutar distintas instancias del programa sobre el mismo fichero concurrentemente). Si se da el caso, se deberá abortar la ejecución de la segunda instancia y escribir en salida de errores "error: another instance of runcmds is running for this commands file: f" (donde f es la ruta del fichero de comandos). Se recomienda usar la llamada al sistema flock(2) para implementar este requisito.

Los comandos se tienen que ejecutar de forma secuencial, en el orden en el que aparecen en el fichero de comandos. Se puede suponer que las líneas del fichero no contienen más de 1023 caracteres. Los comandos descritos en el fichero de comandos pueden tener argumentos. Se puede establecer un límite de 255 argumentos para cada comando.

La salida de todos los comandos, filtrada por la expresión regular indicada, debe escribirse en el fichero f.out, donde f es la ruta del fichero de comandos. En caso de leer de la entrada estándar, se escribirá la salida en el fichero “stdin.out”. Si el fichero existe (p. ej. de una ejecución anterior), se debe truncar. Si no existe, se debe crear. Si no se puede crear/truncar, se debe abortar la ejecución del programa.

Se debe ejecutar cada comando redirigiendo su entrada estándar a /dev/null y su salida de errores a un fichero f.número-de-línea.err, siendo f la ruta del fichero de comandos (en caso de leer de la entrada estándar, se usará "stdin" como f) y número-de-línea la línea correspondiente al comando que se ha ejecutado.

Si estos ficheros existen (p. ej. de una ejecución anterior), se deben truncar. Si no existen, se deben crear. Si no se pueden crear/truncar, la ejecución de ese comando debe fallar.

Los comandos se deben buscar en el directorio actual, /bin y /usr/bin (en ese orden).

El programa debe escribir por su salida el número de la línea y a continuación si ha ejecutado correctamente ("SUCCESS") o ha fallado ("FAIL"). El programa debe intentar ejecutar todos los comandos del fichero.

No se permite ejecutar un shell o usar system(). Se debe usar las llamadas execv(2) / execl(2) para ejecutar programas.

Un posible ejemplo es:

```
$ ls
f1
runcmds
$ cat f1
echo hola
echo fsfasf
comandoqueno no existe
```

```
echo hIla
ls /
$ ./runcmds ‘h.la’ f1
0: SUCCESS
1: SUCCESS
2: FAIL
3: SUCCESS
4: SUCCESS
$ ls
f1
f1.0.err
f1.1.err
f1.2.err
f1.3.err
f1.4.err
f1.out
runcmds
$ cat f1.out
hola
hIla
$ rm f1*
$ ./runcmds ‘.’
echo hola
0: SUCCESS
echo adios
1: SUCCESS
^D
$ ls
runcmds
stdin.0.err
stdin.1.err
stdin.out
$ cat stdin.out
hola
adios
$
```

Runsinfo

Escriba un programa en C runsinfo para GNU/Linux cuyo objetivo es ejecutar el programa showinfo (que existirá en $HOME/bin en el sistema de prueba y que hay que simular) con el entorno y los parámetros adecuados. En el sistema de prueba, el programa showinfo llamará a otros programas que supondrán que el directorio $HOME/bin se encuentra en el PATH. Para que showinfo pueda realizar su trabajo, debe heredar en su entorno una variable PATH que contenga $HOME/bin.

Adicionalmente programa showinfo debe recibir el nombre del usuario actual como parámetro.

Para probar, puedes crear showinfo (que no forma parte de la entrega) como un script:

```
$ echo $HOME
/home/paurea
$ whoami
paurea
$ mkdir $HOME/bin
$ vi $HOME/bin/showinfo
      # se edita ...
$ cat $HOME/bin/showinfo
#!/bin/sh
echo el argumento es $1
echo el path es $PATH

$ chmod u+x showinfo
$ runsinfo
el argumento es paurea
el path es /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/
paurea
```

El programa en C no puede ejecutar ningún comando externo salvo showinfo ni utilizar system.

Runusers

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare runusers.git
$>  git clone runusers.git runusers
$> cd runusers
$> vi runusers.sh
#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf runusers.tgz runusers.git
```

Escribe un script de shell que recibe como parámetros dos parámetros optativos, -q y -p y, a continuación, un conjunto de usuarios (al menos dos).

El script se va a quedar haciendo polling indefinidamente cada 2 segundos. En el momento en el que haya dos usuarios diferentes entre sí incluidos en el conjunto de parámetros ejecutando procesos en la máquina, el programa dejará el resultado de un ps -ef (pero sólo las líneas de los procesos que estén ejecutando un comando que esté en un directorio del PATH) en el fichero log.txt en el mismo directorio de ejecución y saldrá con éxito.

A la hora de ver qué usuarios están, hay que ver cuáles hay ejecutando procesos, no cuáles han hecho login (los que devuelve ps, con los modificadores adecuados, no who).

En caso de que el fichero log.txt exista ya (aunque esté vacío), añadirá al final del fichero y antes de nada, debe escribir una línea con ### fecha ### donde la fecha tendrá como formato algo similar a 02 31 2020 (ver date(1)).

En caso de recibir como parámetro optativo -p, el script escribirá por su salida estándar (y no escribirá las líneas con fecha) en lugar de en el fichero. En caso de recibir como parámetro optativo -q, el script ejecutará sin escribir en el fichero ni en su salida (será silencioso). En ambos casos, esperará a que haya al menos dos usuarios para salir. En caso de recibir ambos modificadores o en caso de haber cualquier otro error, el script debe imprimir un mensaje de error y salir con estado de error.

Este es un posible ejemplo de ejecución:

```
$> ls -l log.txt
ls: cannot access 'log.txt': No such file or directory
#paurea pepe rafa y juan no están
$> runusers paurea pepe juan rafa
..
                            <-#ejecutan cosas paurea y pepe
$> cat log.txt
root      1314     1  0 09:24 ?        00:00:00  /bin/bash
...
paurea    1962  1924  0 09:26 ?        00:00:00   0:00 /bin/sh
...
pepe    2039  1962  0 09:26 ?        00:00:05 /usr/bin/qterminal
pepe    2039  1962  0 09:26 ?        00:00:05 /bin/bash
                            <-#paurea pepe se van
$> runusers paurea pepe juan rafa
```

```
..
                              <-#ejecutan cosas juan y pepe
$> cat log.txt
root      1846     1  0 09:24 ?        00:00:00 /bin/bash
...
paurea    1956   1956 09:26 ?        00:06:13 /usr/bin/inkscape
...
paurea    2042  1962  0 09:26 ?        00:00:00 /usr/bin/xterm
paurea    2044  1962  0 09:26 ?        00:00:15 /bin/bash
### 02 31 2020 ###
root      1216     1  0 09:24 ?        00:00:02 /bin/bash
...
juan     2070     1  0 09:26 ?        00:00:00 /usr/bin/xterm
juan    2078     1  0 09:26 ?        00:00:01 /usr/bin/xterm
...
pepe    2094     1  0 09:26 ?        00:00:00 /bin/bash
pepe    2104     1  0 09:26 ?        00:00:00 /bin/bash
$> runusers -p paurea pepe juan rafa
root      1226     1  0 09:24 ?        00:00:00 /bin/bash
...
juan     1314     1  0 09:24 ?        00:00:00  /usr/bin/xterm
juan        1609  1314  0 09:24 ?        00:00:00 /usr/bin/xterm
...
pepe    2194  1939  0 09:26 ?        00:00:00 /bin/bash
pepe   2452  1846  0 09:26 ?        00:00:00 /bin/bash
$> runusers -q paurea pepe juan rafa
$>
```

El script se debe desarrollar en una única rama. La versión final a entregar estará etiquetada mediante una tag v1.0.0runusers.

El fichero de entrega se debe llamar runusers.tgz

Sha1dir

Escribe un programa en C para Linux llamado sha1dir.c que recibe dos argumentos: la ruta de un directorio y un nombre de fichero de salida.

El programa debe, para todos los ficheros convencionales del directorio, calcular su hash SHA1 con el comando externo sha1sum(1). El fichero de salida debe contener líneas con el inodo, la hash del fichero y su nombre. Las columnas deben ir separadas por espacios (uno o más).

Por ejemplo:

```
$> ls -li
total 20
1046536 drwxr-xr-x 2 esoriano profes 4096 Nov 19 11:02 a
1046535 drwxr-xr-x 2 esoriano profes 4096 Nov 19 11:02 d
1046531 -rw-r--r-- 1 esoriano profes    5 Nov 19 11:02 f1
1046532 -rw-r--r-- 1 esoriano profes    6 Nov 19 11:02 f2
1046533 -rw-r--r-- 1 esoriano profes    0 Nov 19 11:02 f3
1046534 -rw-r--r-- 1 esoriano profes    9 Nov 19 11:02 f4
$> ./sha1dir . out
$> cat out
1046531 63bbfea82b8880ed33cdb762aa11fab722a90a24  f1
1046532 63b76b20dd24f5a4104e9c412be1ba8e71b6b05c  f2
1046533 da39a3ee5e6b4b0d3255bfef95601890afd80709  f3
1046534 5e5e3a6b7d7e4cadc32ebfbb7f7258395f705f72  f4
$>
```

No se puede usar la shell, system(3) o pipes. Hay que duplicar los descriptores de fichero con dup(2) para realizar el ejercicio.

Sha2dir

Escribe un programa en C para Linux llamado sha2dir.c que recibe dos argumentos: la ruta de un directorio y un nombre de fichero de salida.

El programa debe, para todos los ficheros convencionales acabados en txt del directorio, calcular su hash SHA256 con el comando externo sha256sum(1). El fichero de salida debe contener líneas con el inodo, la hash del fichero y su nombre. Las columnas deben ir separadas por espacios (uno o más).

Por ejemplo:

```
$> ls -li
total 20
1046536 drwxr-xr-x 2 esoriano profes 4096 Nov 19 11:02 a
1046535 drwxr-xr-x 2 esoriano profes 4096 Nov 19 11:02 d
1046531 -rw-r--r-- 1 esoriano profes    5 Nov 19 11:02 f1.txt
1046532 -rw-r--r-- 1 esoriano profes    6 Nov 19 11:02 f2.txt
1046533 -rw-r--r-- 1 esoriano profes    0 Nov 19 11:02 f3.txt
1046534 -rw-r--r-- 1 esoriano profes    9 Nov 19 11:02 f4
$> ./sha1dir . out
$> cat out
1046531 80e43222ef09f2305d593a84ba570b5bbbb31e9de90d3e7768ea65a47c6a67d2  f1
1046532 b3a55b9f434a95739a2399b37547cdf936df212275f8aedaec2add5f074ddc27  f2
1046533 0b73add49fc330f50e35c21369abd0383a1ad7d7d4e190acfc8bc64f1d7042fd  f3
$>
```

No se puede usar la shell, system(3) o pipes. Hay que duplicar los descriptores de fichero con dup(2) para realizar el ejercicio.

Shecho

Hay que implementar un script shecho.sh de shell en sh para Linux que actúe de forma similar a echo.

Sin argumentos, escribe sus argumentos por la salida separados por tabuladores. Si un argumento tiene espacios, no lo corta. Con el argumento -s corta los argumentos por cualquier espacio (el IFS

por defecto). Con -h escribe su uso por la salida de error y sale con error. No recibir argumentos no se considera un error.

Por ejemplo:

```
$ ./shecho.sh uno dos tres
uno	dos	tres
$ ./shecho.sh 'uno dos' tres
uno dos	tres
$ ./shecho.sh -s 'uno dos' tres
uno	dos	tres
$ ./shecho.sh -h && echo no se ve
usage: shecho.sh [-s] [arg] …
```

Showvar

Escriba un programa showvar.c en C para Linux que reciba un único parámetro, que debe ser un nombre de una variable de entorno.

Si la variable indicada es una lista de elementos separados por el carácter ':', el programa debe escribir todos los elementos de la variable de entorno por su salida, uno por línea. Si el contenido no se ajusta a lo anterior, debe imprimir el contenido de la variable en una sola línea. En caso de que la variable no exista, debe escribir un error y terminar sin éxito. También debe fallar si no se proporciona ningún argumento o se proporcionan varios.

Para separar los elementos hay que usar la función strtok_r(3).

Por ejemplo:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/
local/games:/usr/lib/go/bin:/opt/puppetlabs/bin/:./
$ ./showvar PATH
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/games
/usr/lib/go/bin
/opt/puppetlabs/bin/
./
$ ./showvar USER
esoriano
$ ./showvar NoExiste
error: variable NoExiste does not exist
./showvar uno dos tres
usage: showvar var
$
$
```

Slay

Escribe un comando slay.c en C para Linux.

El comando recibe un argumento obligatorio que tiene que ser un número diferente de 0. El argumento representa un pid. El comando mandará la señal SIGKILL dicho pid. Si hay cualquier problema con los argumentos, escribirá su uso por la salida de error y saldrá con error.

Si hay algún problema mandando la señal, escribirá la razón por la salida de error y saldrá con error.

Ejemplo:

```
$ slay
usage: slay pid
$ slay 0
usage: slay pid
$ slay hola
usage: slay pid

#el proceso 32 es de root
$ slay 32
Operation not permitted
$ echo $?
1
$ sleep 20
[1] 2059
$ slay 2056
$ echo $?
0
[1]+  Killed                 sleep 20
```

Sole

Escribe un programa en C para Linux sole.c que acepte como argumentos una serie de ficheros. El programa los comparará con cmp -s. Para cada fichero que reciba como parámetro, se debe debe tener un hijo que lo compare con los demás. Esos procesos deben ejecutar de forma concurrente.

La salida del programa será la lista de ficheros, en el orden en el que se pasaron en la invocación, con "yes" o "no" dependiendo de si el fichero es único (esto es, distinto a todos los demás) o no lo es:

```
$> echo eq > b
$> cp b d
$> echo 1 > a
$> echo 2 > c
$> sole a b c d
a yes
b no
c yes
d no
$>
```

Sortstr

Escribe un programa sortstr.c en C para Linux que reciba como argumentos palabras (un número indeterminado mayor que 0) y escriba por su salida la lista de palabras ordenada. El programa debe ignorar las palabras que no tengan ninguna vocal.

Para ordenar, se debe implementar el algoritmo insertion sort (https://en.wikipedia.org/wiki/Insertion_sort).

Ejemplo:

```
$> ./sortstr  ffhfhhfhf hola adios pepe rrrrr pepin pepe
adios
hola
pepe
pepe
pepin
$>
```

Sourcefiles

Escriba un programa en C llamado sourcefiles.c que lea líneas de su entrada estándar con paths. Por cada path, el programa debe escribir a continuación el número de ficheros fuente que ha encontrado en ese path (en ese directorio y todos los subdirectorios que contiene, esto es, lo debe hacer recursivamente). Por cada path debe escribir:

    El path
    Número de ficheros .c encontrados en ese path
    Número de ficheros .h encontrados en ese path
    Suma del tamaño en bytes de todos los ficheros
      .c y .h que ha encontrado en ese path

Si procesando algún directorio se encuentran errores, se debe notificar como corresponda, pero se deben seguir procesando líneas hasta el final de la entrada. Si hay algún problema, al final el programa deberá salir con estatus de fallo.

La salida se tiene que ajustar a la del ejemplo:

```
$> cat paths
/home/pepe
/home/manuel
$> ./sourcefiles < paths
/home/pepe    12    2   33124
/home/manuel  123   14  4234254
$>
```

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare spooldir.git
$>  git clone spooldir.git spooldir
$> cd spooldir
$> vi spooldir.sh

#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf spooldir.tgz spooldir.git
```

Spooldir

Escribe un script de shell llamado spooldir.sh que reciba dos parámetros obligatorios: un directorio que tiene que existir y un nombre de comando Cmd.

El script se debe quedar esperando (haciendo polling cada 10 segundos) vigilando el directorio para ver si se crea algún fichero.
Si se crea un fichero que se llama FINISH, el script debe acabar. Si los ficheros creados no se llaman FINISH, el script debe ir ejecutando el comando Cmd con el
nombre de cada fichero recientemente creado como argumento, esperar a que acabe y si todavía existe el fichero, renombrarlo acabado en .done.
Si el fichero ya acababa en .done no es necesario ejecutar el comando sobre él.

Una segunda versión del script debe hacer exactamente lo mismo que la primera y adicionalmente borrar los ficheros .done del directorio antiguos (los que tengan más de una hora de vida).

Una tercera versión, además, recibe como primer parámetro optativo el nombre de fichero que hace que salga (que era FINISH en la otra versión).

En el repositorio debe haber dos ramas, una master con una con la versión básica que no borra ficheros y requiere que el
fichero se llame FINISH (etiquetada con la tag v0.0.1ro) y otra descendiente de esta versión que se bifurca justo a partir de la etiqueta. Esta segunda versión sí borra (con la tag v0.0.1w).

Debe haber un merge entre ambas que continúe en la rama master y que contenga la tercera versión con todo v0.1.0.

PISTA: Para ver si se crea un fichero, se puede salvar el resultado del listado en un fichero temporal. Para evitar colisiones puedes usar el comando mktemp(1).

Una idea del aspecto del grafo de commits del repositorio (puede haber más o menos de los commits representados por bolas naranjas, no tienen que coincidir uno a uno):

Terre

Escribe un script de shell llamado terre.sh que obtenga estadísticas a partir de un fichero de datos sismológicos. El fichero contendrá datos relativos a diferentes regiones del planeta. Cada región va precedida de una cabecera con el nombre de la región. El nombre siempre comienza a principio de la línea. Las líneas vacía y los comentarios, que serán líneas comenzando por el carácter '#' , se deben ignorar. Después del nombre, vienen líneas con 4 columnas con números en coma flotante separadas por tabuladores:

```
Latitud
      Longitud
      Profundidad (km)
      Magnitud (mbLg)
```

La salida debe ser, para cada región una línea con el nombre de la región todo en minúsculas, la profundidad media y la magnitud máxima.

Por ejemplo:

```
$> cat datos.txt
#Latitud    Longitud    Profundidad (km)  Magnitud (mbLg)
TARRAGONA
41.2985     0.9995      12.00 2.2
41.3097     0.9929      2.02  1.9
41.1512     1.1632      12.00 2.0

GRANADA
37.4084     -3.6169     13.2  1.6
37.4083     -3.6162     13.1  1.52
$> ./terre.sh datos.txt
tarragona 8.6733 2.2
granada 13.15 1.6
$>
```

Escriba un programa en C testcoms para GNU/Linux que reciba dos parámetros y un tercero opcional.

El primer argumento, N, es el número total de veces que se ejecutará el segundo argumento, que es el nombre de un comando.

El objetivo es probar el comando ejecutándolo repetidamente, pasándole un número pseudoaleatorio diferente como único argumento en cada ejecución.

El tercer argumento, que es opcional, define M. M indica las copias del comando que se ejecutarán concurrentemente. Si se proporciona este argumento, el programa ejecutará el comando en M procesos concurrentes y esperará a que acaben todas antes de ejecutar los siguientes M concurrentes. Hay que tener en cuenta que el último grupo puede ser menor que M, si M no es divisor de N. Si no se recibe el tercer argumento, M tendrá un valor por defecto de 1.

Si algún comando falla, en el momento de que se detecte, se escribirá por la salida la llamada que ha hecho que ejecute con error precedida de 'error in command: ' y se acabará el programa.

El programa completo debe salir con éxito sólo si las N copias han salido con éxito.

Por ejemplo:

```
$ ./testcoms
usage: testcoms N cmd [M]
$ ./testcoms 3 xy 4 5
usage: testcoms N cmd [M]
$ cat prueba.sh
#!/bin/sh
echo empieza $1
sleep 10
echo acaba $1
$ ./testcoms 5 ./prueba.sh 2
echo empieza 1804289383
echo empieza 846930886
                        #pasan 10 segundos
echo acaba 1804289383
echo acaba 846930886
echo empieza 1681692777
echo empieza 1714636915
                        #pasan 10 segundos
echo acaba 1714636915
echo acaba 1681692777
echo empieza 1714636915
                        #pasan 10 segundos
echo acaba 1714636915
$echo $?
0
$ ./testcoms 5 ./prueba.sh
echo empieza 1804289383
                        #pasan 10 segundos
echo acaba 1804289383
echo empieza 846930886
                        #pasan 10 segundos
echo acaba 846930886
echo empieza 1681692777
                        #pasan 10 segundos
echo acaba 1681692777
echo empieza 1714636915
                        #pasan 10 segundos
echo acaba 1714636915
echo empieza 1957747793
                        #pasan 10 segundos
echo acaba 1957747793
$echo $?
0
$ ./testcoms 2 false 5
error in command: false 846930886
$ echo $?
1
```

El programa en C debe ejecutar los programas con la llamada al sistema execv(2). Para generar números pseudoaleatorios, hay que utilizar rand(3), con semilla 1 (el valor por defecto). No es necesario forzar ningún rango concreto, el valor devuelto por rand(3) vale.

Trlite

Escriba un programa en C para Linux que reciba dos parámetros obligatorios que sean dos caracteres. El programa debe leer lineas de la entrada estándar, y por cada línea leída, debe sustituir el carácter indicado por el primer argumento por el carácter indicado en el segundo argumento. Después, debe escribir por su salida el número de línea seguido de la línea con el carácter sustituido. Se puede suponer que el tamaño máximo de una línea es 512 bytes.

El formato de la salida debe ser exactamente el mismo que el del siguiente ejemplo:

```
$> cat fichero
esto es una linea
esta es otra linea
adios
$> ./trlite a A < fichero
0: esto es unA lineA
1: estA es otrA lineA
2: Adios
$>
```

El programa debe funcionar con ficheros de gran tamaño.
El programa debe escribir errores de la forma adecuada y salir con el status correspondiente si no recibe los parámetros adecuados, o sucede cualquier error.
El programa consistirá de un único fichero: trlite.c.

Txtsha2

Escriba en C para Linux un programa llamado txtsha2.c cuyo propósito es crear un resumen hash SHA-256 del resultado de
la concatenación de todos los ficheros regulares acabados en .txt de un directorio (ojo, no tienen por qué contener texto).

El orden de la concatenación debe ser el orden en el que se encuentran en el directorio (no hay que ordenar los ficheros de ninguna forma). El programa sólo admite un argumento para indicar el directorio sobre el que se quiere trabajar. Si no se le pasa ningún argumento, debe actuar sobre el directorio de trabajo actual.

El resumen SHA-256 debe calcularse ejecutando el comando sha256sum(1), que debe ejecutarse sin ningún argumento. Este comando lee su entrada estándar hasta que se acaba, para luego calcular el resumen SHA-256 de los datos leídos y escribirlo por su salida estándar.

La salida del programa tiene que ser, simplemente, los 32 bytes expresados en hexadecimal del resumen SHA-256 generado. No se puede escribir nada más, ni antes ni después del SHA-256. En particular, los datos extra que escribe el comando sha1sum a su salida (espacios, guión...), no deben aparecer a la salida del programa.

No se permite la ejecución de un shell ni el uso de la llamada system().

A continuación se muestra un ejemplo:

```
$> txtsha2 /tmp/d
3ee41b364227bcea0b8ee704d718a27da932062cd23fdd851dbf54c8c6181f13
$>
```

Uniqfiles

Escriba un programa en C para Linux llamado uniqfiles.c que, dada una lista de nombres de ficheros, los compare dos a dos con el comando cmp y escriba por su salida los ficheros que son únicos (que no hay otro fichero igual en la lista).

Si hay ficheros iguales (al menos dos), debe salir con estatus de fallo, en otro caso con éxito. Si no se le pasa ningún fichero, tiene que salir con éxito. Si cmp falla (status de salida 2), debemos escribir cómo se usa el comando y nada más.

La opción -s de cmp suprime los mensajes de error.

El programa debe ser lo más concurrente posible.

Ejemplo:

```
$> ./uniqfiles
$> ./uniqfiles noexiste noexiste
usage: uniqfiles [files ...]
$> cat a
hola
$> cat b
hola
$> cat /tmp/c
adios
$> cat d
pepe
$> ./uniqfiles a b /tmp/c ./d
/tmp/c
./d
$>
```

Entregue un fichero uniqfiles.c con el fuente del programa.

Usbpower

Clona el siguiente repositorio de Git:

https://gitlab.etsit.urjc.es/paurea/usbpower.git

Completa el script usbpower.sh. Dicho script debe sacar un listado de todos los dispositivos USB que se han identificado, (dmesg proporciona dicha información). Cada dispositivo USB se identifica con dos números en hexadecimal, el identificador de vendor y el identificador de producto. Para cada dispositivo, se debe escribir (con el formato exacto descrito a continuación) el vendor, el producto y la corriente máxima requerida (esa información la proporciona el comando lsusb). Al final, se debe escribir la corriente total requerida por todos los dispositivos (la suma de todos).

Por ejemplo:

```
$> usbpower.sh
VENDOR:43ED ID:4312 MAXPOWER:0mA
VENDOR:DD32 ID:009C MAXPOWER:2mA
VENDOR:E531 ID:E89F MAXPOWER:100mA
VENDOR:3121 ID:8AC4 MAXPOWER:0mA
VENDOR:4324 ID:E78A MAXPOWER:199mA
VENDOR:E43F ID:2342 MAXPOWER:NA
TOTAL POWER: 301mA
$>
```

Nótese que los dígitos hexa están en mayúsculas y que siempre son 4 dígitos.

Puede haber dispositivos para los que no podamos conseguir su corriente (o bien porque no lo proporciona el comando lsusb o porque no se pueda conseguir por problemas de permisos). En estos casos, se deberá escribir el valor NA (ver el ejemplo) y no se debe escribir nada por la salida de errores.

Si para un dispositivo hay varias corrientes máximas en la salida de lsusb, se debe elegir siempre la primera corriente máxima.

Se debe entregar un fichero usbpower.tgz con el proyecto git completo. El repositorio entregado puede tener todos los commits que se desee. El git tiene que tener la versión final del script con una tag con nombre release1.0.0

Users

Escribe un script de shell users.sh que para cada usuario que está trabajando ahora mismo en el sistema, proporcione la siguiente información:

- Nombre y apellidos, junto con su login
- Path de su directorio home
- Día más lejano en el tiempo en el que ha entrado en el sistema según la información proporcionada por el comando last (que las ordena por fecha)
- Si tiene ejecutando algún programa de python (suponiendo que el interprete de python puede estar en cualquier directorio, por ejemplo /usr/sbin/python3 o /home/pepe/anaconda3/python)

La información se debe proporcionar exactamente con el formato exacto del siguiente ejemplo (mayúsculas, etc.):

```
$> who
pepon  pts/0        2019-04-26 11:41 (212.128.254.157)
pepa   pts/1        2019-04-26 11:54 (193.147.79.100)
$> users.sh
JOSE PEREZ RAMIREZ (pepon)
HOME: /home/alumnos/pepe
OLDEST SESSION:  Sun Apr  7
EXECUTING PYTHON: yes

JOSEFA ROMERO GARCIA (pepa)
HOME: /home/pepa
OLDEST SESSION: Tue Apr  9
EXECUTING PYTHON: no

$>
```

Se recomienda la lectura de las páginas de manual de los comandos who, finger y ps.

Waitexit

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio de git. Entregarás un tgz de este directorio:

```
$> cd /work
$>  git init --bare waitexit.git
$>  git clone waitexit.git waitexit
$> cd waitexit
$> vi waitexit.sh

#hago el script y demás, hago push...
$>  cd /work
$> tar -czvf waitexit.tgz waitexit.git
```

Escribe un script de shell llamado waitexit.sh que reciba como parámetro obligatorio un nombre de comando y una lista de pids (al menos uno).
Se debe quedar esperando (haciendo polling cada segundo) hasta que todos los procesos asociados a los pids que se han pasado como
parámetro salgan en cuyo caso debe ejecutar el comando. Si han pasado 20 segundos y no han salido, debe matarlos y no ejecutar el comando.
Para matarlos hay que mandarles la señal -KILL (man kill).

En el repositorio debe haber dos ramas, una con una versión que mande la señal KILL y otra que mande TERM. La versión definitiva de KILL debe estar etiquetada v1.0.0KILL y la otra v1.0.0TERM.
Implementar un script waitrun.sh de shell en sh para Linux que ejecute un comando que

recibe como único parámetro obligatorio (junto con sus argumentos) cuando vea que aparece el fichero /tmp/go. El comando se quedará esperando

durante intervalos de 1 segundo hasta que aparezca dicho fichero.

```
$ ./waitrun.sh 'echo hola' &
$ touch /tmp/go
hola
$ ./waitrun.sh
usage: waitrun.sh cmd
```

El desarrollo de esta práctica lo vas a hacer en git. Crea un repositorio bare de git y un clon a través del sistema de ficheros.
Entregarás un tgz del repositorio bare con todos los commits.

```
$> cd $HOME/work
$>  git init --bare waste.git
$>  git clone waste.git waste
$> cd waste
$> vi waste.sh

#hago los scripts y demás, hago push...
$>  cd $HOME/work
$> tar -czvf waste.tgz waste.git
```

Escribe dos scripts de shell con el mismo nombre, waste.sh.
Ambos scripts se deben desarrollar en ramas separadas (uno puede estar en

master).

- Primera versión:

El primer script imprimirá una lista con los N usuarios que más CPU utilicen (la reportada por ejemplo por ps -A -o pmem,pcpu,user).
El segundo script mostrará los hasta N usuarios que más memoria utilicen (reportada por ps).
Ambos recibirán un único argumento obligatorios, el número de usuarios. En caso de no recibirlo, imprimirá un error por la salida de error y saldrán con error.
Esta versión estará etiquetada como v.0.0.0waste con un tag en ambas ramas.

- Segunda versión:

Todavía en ramas separadas se escribirán ambos scripts para que reciban su parámetro de forma
compatible entre sí. El que mira el uso de cpu recibirá -c N y el que mira el uso de memoria
recibirá -m N ambos parámetros serán obligatorios (la flag y el número).
Esta versión estará etiquetada como v.0.1.0waste con un tag en ambas ramas.

- Tercera versión:

Se hará merge de ambas ramas a máster y se dejará un único script que reciba obligatoriamente -c N o -m N
y en tal caso actúe como el script adecuado de la versión anterior.
Esta versión estará etiquetada como v.1.0.0waste con un tag en la rama máster.

Ejemplo:

```
$ git checkout v.1.0.0waste
$ ./waste.sh -c
usage:  ./waste.sh [-c N|-m N]
$ ./waste.sh
usage:  ./waste.sh [-c N|-m N]
$ ./waste.sh -c 5
paurea
root
rtkit
syslog
systemd-resolve
$ ./waste.sh -m 3
paurea
root
rtkit
```

Escribe un programa llamado xpaths.c en C para linux que reciba un conjunto de ficheros,
que deben contener dentro paths (uno por línea). El programa debe encontrar el path más común (el que más se repite en esos ficheros) y utilizarlo como salida estándar del comando ps. El fichero debe truncarse.

Si hay algún error obteniendo el fichero más común, la salida de ps se debe escribir por la salida de errores. Se considera un error:

 - Que no exista o no sea accesible algún fichero pasado como argumento.
 - Que no exista alguno de los paths contenidos en los ficheros.

No es un error que no haya ningún path en los ficheros (o que no haya parámetros).
En ese caso, la salida de ps se tiene que escribir en la salida estándar.

Ejemplos:

```
$> cat uno
/tmp/a
/tmp/b
$> cat dos
/tmp/b
$> cat tres
/tmp/c
/tmp/b
/tmp/b
$> xpaths uno dos tres
$> cat /tmp/b
  PID TTY          TIME CMD
 5355 pts/0    00:00:00 bash
 5390 pts/0    00:00:00 atom
 5497 pts/0    00:00:00 atom
15111 pts/0    00:00:12 xpaths
15140 pts/0    00:00:00 ps
$> xpaths
  PID TTY          TIME CMD
 5355 pts/0    00:00:00 bash
 5390 pts/0    00:00:00 atom
 5497 pts/0    00:00:00 atom
16112 pts/0    00:00:12 xpaths
16144 pts/0    00:00:00 ps
$> xpaths /noexiste 2> /tmp/errs
$> cat /tmp/errs
  PID TTY          TIME CMD
 5355 pts/0    00:00:00 bash
 5390 pts/0    00:00:00 atom
 5437 pts/0    00:00:26 atom
17154 pts/0    00:00:12 xpaths
17187 pts/0    00:00:00 ps
```

Zcount

Escriba un programa en C para Linux zcount.c cuente el número de bytes a cero en cada uno de los ficheros de un directorio y escriba en el mismo directorio un fichero z.txt con una línea por fichero. Cada línea contendrá el número de bytes a cero, un tabulador y el nombre del fichero. El programa debe recibir un solo parámetro que será el directorio a procesar. Si el fichero z.txt existe, debe ignorarlo para la cuenta, truncarlo y sobreescribirlo.

Ej:

```
$> mkdir /tmp/a
$> echo aaa> /tmp/a/nada
$> dd if=/dev/zero of=/tmp/a/megas bs=10M count=1
1+0 records in
1+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0,0169875 s, 617 MB/s
$> dd if=/dev/zero of=/tmp/a/diez bs=10 count=1
1+0 records in
1+0 records out
10 bytes copied, 0,000119917 s, 83,4 kB/s
$> echo bbbbb >> /tmp/a/diez
$> ls /tmp/a
diez  megas  nada
$> zcount /tmp/a
$> ls /tmp/a
diez  megas  nada   z.txt
$> cat /tmp/z.txt
10 diez
10485760 megas
0 nada
```

Para leer, se debe usar la llamada al sistema read. Para este ejercicio no está permitido usar las funciones con buffering de stdio (fread, etc).
Escribe un comando similar a tee(1) pero que descomprima los datos que lee de la entrada estándar. El fichero fuente se debe llamar ztee.c.

Una invocación sería como sigue:

```
$>  echo aaaa > zzz
$>  gzip zzz
$>  ztee fichsalida < zzz.gz
aaaa
$> cat fichsalida
aaaa
$>
```

Este comando lee de su entrada estándar datos comprimidos con gzip sin ningún argumento y los descomprime mediante el comando gunzip. El fichero de salida se debe crear o truncar si existe.

El programa debe escribir lo mismo (es decir, el resultado de descomprimir los datos de su entrada estándar) en su salida estándar y en el fichero que recibe como parámetro.

Naturalmente, no se permite la utilización del comando tee ni ejecutar un shell.