Makefile (0,5 puntos)

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

Control de errores (0,5 puntos)

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

Ejercicio 1 (2,5 puntos)

Haz un programa llamado **mostrarJP.c** para mostrar por pantalla una jerarquía de procesos en ejecución. El programa debe recibir un argumento, *PID*, que corresponderá a un identificador de proceso, y debe comprobar que sea válido, es decir, que sea un número[1] más grande que 0 y más pequeño que el valor guardado en el fichero */proc/sys/kernel/pid_max* (consulta este valor desde el intérprete de comandos y defínelo en tu código como una constante) y que el proceso correspondiente exista[2]. Utiliza la función `assert` (consulta el manual para ver su funcionamiento) para comprobar el PID es válido. Si es el caso, el programa *mostrarJP* creará un proceso hijo que se encargará de mostrar por pantalla la jerarquía de procesos que tiene como raíz al proceso *PID* mediante la mutación de su código a "`pstree -p <PID>`". El programa debe esperar al proceso hijo (pero sin recoger su código de finalización).

Ejemplos de salida:

```
$ ./mostrarJP $$
bash(5871)──mostrarJP(6028)──pstree(6029)
$ ./mostrarJP
mostrarJP: mostrarJP.c:21: main: Assertion `argc == 2' failed.
Aborted (core dumped)$
$ ./mostrarJP 0
mostrarJP: mostrarJP.c:24: main: Assertion `(pid > 0) && (pid < 4194304)' failed.
Aborted (core dumped)
$ ./mostrarJP 9999
mostrarJP: mostrarJP.c:26: main: Assertion `errno != ESRCH' failed.
Aborted (core dumped)
```

ATENCIÓN! Este ejercicio se evaluará en función de la ejecución correcta del programa en los diferentes casos: 1,5 puntos para ejecuciones con parámetros correctos (mira el primer ejemplo), 1 punto para ejecuciones con parámetros incorrectos (mira el resto de ejemplos). Si la ejecución no da el resultado esperado, el código no se evaluará.

Ejercicio 2 (4 puntos)

Haz un programa llamado **crearJP.c** para crear una jerarquía de procesos. Este programa debe recibir dos argumentos. El primero, *nhijos1*, será un número con valor entre 1 y 10 (ambos

[1] Recomendamos usar la función `strtol` en lugar de `atoi` para convertir cadenas de caracteres a enteros, ya que facilita mucho la gestión de errores cuando la cadena de caracteres no es un número válido.

[2] Puedes comprobar si la invocación `kill(pid,0)` retorna un error `ESRCH` para verificar si un pid existe.

incluidos) que corresponderá con la cantidad de procesos hijo en el primer nivel de la jerarquía (es decir, hijos directos del proceso que ejecuta inicialmente el programa *crearJP*). El segundo, *nhijos2*, será un número con valor entre 1 y 5 (ambos incluidos) que corresponderá con la cantidad de procesos hijo en el segundo nivel de la jerarquía para cada uno de los procesos del nivel anterior. Después de comprobar los parámetros, el programa *crearJP* escribirá por pantalla un mensaje con su PID y creará de manera concurrente tantos procesos como indique el parámetro *nhijos1*. Por su parte, cada proceso hijo creará de manera concurrente tantos procesos como indique el parámetro *nhijos2*. Por otro lado, después de crear los procesos hijos de primer nivel, el proceso inicial del programa *crearJP* debe configurar una alarma con una duración de 60 segundos, y esperar que salte esta alarma sin consumir CPU. En este momento, el programa enviará un signal SIGUSR1 a los procesos hijos del primer nivel de la jerarquía. En cualquier momento, después de ejecutar *crearJP,* los procesos hijos de ambos niveles pueden recibir un SIGUSR*2* desde el terminal. Por su parte, cada proceso hijo reenviará el signal que reciba a sus procesos hijos o, si no tiene hijos, a sus hermanos mayores. Para hacerlo, deberá haber reprogramado los signals SIGUSR1 y SIGUSR2 de manera que tengan la misma rutina de tratamiento y que el tratamiento vuelva al comportamiento por defecto después de recibir el primer signal. Todos los procesos hijos de ambos niveles deberán esperar la recepción del signal SIGUSR1 o SIGUSR2 sin consumir CPU. Cuando reciban el signal, todos los procesos hijos escribirán un mensaje por la pantalla con su PID, el PID de su padre, y qué signal han recibido (SIGUSR1 o SIGUSR2)[3] y acaban. Implementa una solución en que este mensaje no se escriba desde la rutina de atención al signal. Añade también el bloqueo de signals que sea necesario en las máscaras de los procesos para asegurar que no se pierde ninguno. Finalmente, recuerda que todos los procesos deben esperar a sus procesos hijos respectivos (sin recoger su código de finalización) antes de acabar.

Ejemplos de salida (al último, 6270 ha recibido un USR2 desde el terminal):

```
$ ./crearJP 3 2
Soy el proceso 6207, raiz de la jerarquia
Soy el proceso 6208 (hijo del proceso 6207):  Recibido signal User defined signal 1 (10)
Soy el proceso 6209 (hijo del proceso 6207):  Recibido signal User defined signal 1 (10)
Soy el proceso 6210 (hijo del proceso 6207):  Recibido signal User defined signal 1 (10)
Soy el proceso 6215 (hijo del proceso 6208):  Recibido signal User defined signal 1 (10)
Soy el proceso 6216 (hijo del proceso 6208):  Recibido signal User defined signal 1 (10)
Soy el proceso 6212 (hijo del proceso 6209):  Recibido signal User defined signal 1 (10)
Soy el proceso 6213 (hijo del proceso 6209):  Recibido signal User defined signal 1 (10)
Soy el proceso 6211 (hijo del proceso 6210):  Recibido signal User defined signal 1 (10)
Soy el proceso 6214 (hijo del proceso 6210):  Recibido signal User defined signal 1 (10)

$ ./crearJP 3
Usage: crearJP <NFILLS1> <NFILLS2>, 1 =< NFILLS1 =< 10, 1 =< NFILLS2 =< 5

$ ./crearJP 5 7
Usage: crearJP <NFILLS1> <NFILLS2>, 1 =< NFILLS1 =< 10, 1 =< NFILLS2 =< 5

$ ./crearJP 1 3
Soc el proces 6267, arrel de la jerarquia
Soc el proces 6270 (creat pel proces 6268):  Acabo per signal User defined signal 2 (12)
Soc el proces 6269 (creat pel proces 6268):  Acabo per signal User defined signal 2 (12)
Soc el proces 6268 (creat pel proces 6267):  Acabo per signal User defined signal 1 (10)
Soc el proces 6271 (creat pel proces 6268):  Acabo per signal User defined signal 1 (10)
```

Ejercicio 3 (2,5 puntos)

Haz un programa llamado **mostrarNJPs.c** para mostrar por pantalla una o varias jerarquías de procesos en ejecución. El programa puede recibir uno o más argumentos, *PID1 [PID2 ... PIDN]*, que corresponderán a los identificadores de los procesos raíz de cada una de las jerarquías. Para

[3] Podéis utilizar la función `strsignal(s)`.

cada PID, el programa deberá crear un nuevo proceso y mutarlo al programa *mostrarJP*[4], pasándole el argumento correspondiente. Los procesos deben ejecutarse de manera concurrente. Para esperar la finalización de los procesos hijos, el padre debe capturar el signal SIGCHLD, y mantenerse en una espera activa mientras no acaben todos los hijos. La rutina de atención al signal SIGCHLD debe recoger el estado de finalización (proceso acabado por exit o por signal) y el valor de finalización (código del exit o número de signal) de cada hijo y escribir esta información por pantalla.

Ejemplo de salida si ejecutamos en un terminal *./crearJP 4 2 & ./crearJP 3 2*, y en otro:

```
$ ./mostrarNJPs 17440 17441 22222
mostrarJP: mostrarJP.c:30: main: Assertion `(errno != ESRCH)' failed.
crearJP(17440)─┬─crearJP(17442)─┬─crearJP(17448)
               │                └─crearJP(17450)
               ├─crearJP(17443)─┬─crearJP(17452)
               │                └─crearJP(17454)
               ├─crearJP(17444)─┬─crearJP(17456)
               │                └─crearJP(17458)
               └─crearJP(17445)─┬─crearJP(17459)
                                └─crearJP(17460)
El procés 17485 ha acabat per exit amb codi 0
crearJP(17441)─┬─crearJP(17446)─┬─crearJP(17461)
               │                └─crearJP(17462)
               ├─crearJP(17447)─┬─crearJP(17455)
               │                └─crearJP(17457)
               └─crearJP(17449)─┬─crearJP(17451)
                                └─crearJP(17453)
El procés 17486 ha acabat per exit amb codi 0
El procés 17487 ha acabat per signal 6
```

Qué hay que hacer

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

Qué se valora

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

Qué hay que entregar

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

```
tar zcvf clab1.tar.gz Makefile *.c
```

[4] Puedes usar el programa *mostrarJP_ok* que se proporciona junto con el enunciado si el primer ejercicio no te ha funcionado.