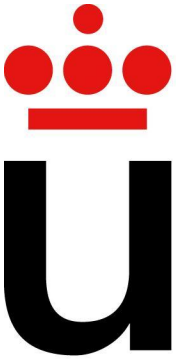

Universidad
Rey Juan Carlos

Sistemas Operativos

[Práctica 2 - MINISHELL]

José Víctor García Llorente

Tabla de contenido

Autor

Nombre: José Víctor.
Apellidos: García Llorente.
Curso: Tercero.
Titulación: Doble grado en ingeniería del software + matemáticas.

Descripción del Código

Diseño del Código

- ALGORITMOS UTILIZADOS

El programa comienza ejecutando la función main, la cual en primera instancia imprime el prompt por primera vez y se manejan las señales necesarias (las señales se explican en el apartado *implementación de señales*) para comenzar a leer líneas por entrada estándar. A continuación se usa la librería parser para extraer las características de la línea que se ha introducido. En primer lugar, descartamos que la línea sea vacía. Si es así, el programa continúa. Después, dependiendo del número de mandatos introducidos, trataremos la línea de una forma u otra.

Si se trata de una línea de un solo mandato, primero comprobamos que la instrucción introducida no sea ni “cd” ni “umask”. Las funciones implementadas *execcd()* y *execumask()* ya comprueban en su interior que el mandato introducido se corresponde con el mandato *cd* o *umask* respectivamente. Si así fuese, dichas funciones ejecutarían el mandato desde el proceso padre. Si no fuese así, se crearía un hijo y dicho hijo ejecutaría el mandato correspondiente (o daría error en caso de que no existiera). Previamente se debe comprobar también que no se ha llamado a *exit*. Por tanto, antes de crear el hijo y ejecutar el mandato (con la llamada a la función *execmandato()*) se debe comprobar si se ha llamado a exit, en cuyo caso la Minishell acabaría con todo proceso y se saldría del programa.

Si se trata de una línea con varios mandatos, se llama a la función *execnmandatos()*, que es la encargada de crear los pipes y los procesos necesarios para ejecutar los n mandatos introducidos y concatenarlos. Ya en su interior se comprueba que el mandato introducido no sea ni *cd* ni *umask*, ya que ambos solo se deben ejecutar si se les llama individualmente. Por último se vuelve a imprimir el prompt y el programa se prepara para la espera de una nueva línea.

Esta es la estructura del programa a grandes rasgos. En las siguientes secciones se explicará con más detalle las funciones implementadas del programa, la estrategia usada y el uso de pipes.

- ESTRATEGIA DE EJECUCIÓN DE MANDATOS Y GESTIÓN DE TUBERÍAS

Como ya se ha explicado, la función *main* es la encargada de distinguir (y en consecuencia, tratar) líneas de acuerdo a si tienen solo uno o varios mandatos. Se ha seguido esta estrategia principalmente para una mejor legibilidad a la hora de entender el código. En esta sección explicaremos las funciones *execmandato* y *execnmandatos* que son las responsables de la gestión de procesos y tuberías.

execmandato: Se crea un solo proceso, encargado de ejecutar el mandato introducido por pantalla. El hijo se encarga de las redirecciones en primer lugar (ver sección de *redirecciones*). Para ello se llama a la función *redirect*. Si se devuelve algún error, entonces se imprime por la salida de error establecida el correspondiente mensaje, pero si no ocurre ningún problema se ejecuta la línea *execv* y el hijo termina. Mientras tanto, al padre se le exige esperara por su hijo.

execnmandatos: La novedad en esta función es que se declara una variable nueva, llamada *arraypipes*. Como su nombre indica, es un array de arrays de dos posiciones (array de pipes). Por lo tanto, nada más comenzar se reserva la memoria suficiente para n-1 pipes (donde n es el número de mandatos

introducidos) y se inicializan. A continuación, es importante establecer la redirección de error estándar en el proceso padre, para que si ocurre algún error, se impriman cada uno de los errores en el fichero correspondiente porque si redirigimos el error en los procesos hijo, como se abre un descriptor de fichero distinto cada vez, los errores se van sobreescribiendo y por lo tanto solo se imprimiría el último.

En *execnmandatos*, al crear cada proceso hijo se deben comprobar varias cosas: en primer lugar, que el mandato introducido no sea ni *cd* ni *umask,* en cuyo caso dichas funciones se encargan de imprimir el mensaje de error. Además, hay que distinguir 3 casos en función del número de hijo en el que nos encontremos:

Si estamos en el hijo o mandato 0 (el primero de todos), Hay que redirigir la entrada estándar al fichero que se demande y comprobar que no hayan ocurrido errores, por supuesto. Posteriormente se debe enviar el resultado de la ejecución del mandato al pipe para que el hijo 1 (segundo hijo) trabaje a partir de ello. Por eso se redirige al pipe 0 la salida estándar.

Si estamos en el hijo i donde 0<i<n-1: Recibimos por entrada lo que haya escrito el hijo anterior por el pipe i-1 y se debe escribir por el pipe i el resultado de la ejecución del comando, para mandarselo al hijo siguiente. Este proceso se repite hasta llegar al último hijo.

Si estamos en el hijo i = n-1: Debemos leer lo que haya escrito el hijo anterior por el pipe i-1 y redirigir la salida estándar al fichero correspondiente.

- IMPLEMENTACIÓN DE BACKGROUND

Cuando se introduce un mandato con intención de ejecutarlo en background, lo primero que hace el proceso padre es guardar el nombre de la línea en el array de procesos. Después (si no se ha llamado a cd, umask, jobs o fg) o bien se ejecutará un único mandato o bien se ejecutarán varios separados por pipes. En cualquiera de los dos casos, al llegar a ejecutarse después del fork el proceso padre, si el mandato ha sido ejecutado en background, se esperará por el hijo correspondiente, pero sin bloquear el programa (con WNOHANG). De esta forma se continuarán ejecutando mandatos, pero siempre añadiendo dicha línea al array de procesos (se guarda el pid clave y el status actual del proceso). Acto seguido se procede a incrementar la variable *max* que indica la nueva posición en la que guardaremos el siguiente proceso que ejecutaremos en background y lo pondremos módulo 1024, que aunque en realidad no sea necesario, porque en principio el array solo estaba pensado para guardar 1024 procesos, siempre es una buena práctica.

Además contamos con dos funciones muy importantes ya que la primera antes de iniciar a ejecutar una nueva línea actualiza el estado de cada proceso en la lista de procesos en background y la segunda muestra en cada iteración y tras ejecutar el mandato o conjunto de mandatos correspondientes si hay procesos en background que han terminado y se pueden eliminar de la lista de procesos. Son las funciónes *updatebackgroundstatus* y *mostrarprocesosterminados.*

- IMPLEMENTACIÓN DE SEÑALES

Uno de los requisitos que debía tener la práctica era que se ignorara la señal SIGINT para salir del programa. Para llevar a cabo dicha tarea, se ha implementado el manejador llamado *abortar*, que imprime por pantalla un salto de línea cuando estamos en mitad de un proceso y terminando la ejecución del mismo. Después de dicho salto de línea se vuelve a mostrar el prompt de nuevo, pero solo si estamos en mitad de

ejecución de un mandato. Si no estamos en mitad de ejecución, entonces significa que hemos llamado a la señal SIGINT (Ctrl+C) mientras se pedía una nueva línea (durante el *fgets*). En este caso, para evitar la salida de la minishell, se ignora la señal, pero no se puede reprogramar para que vuelva a imprimir el prompt ya que nada más ejecutar el manejador se volvería al punto donde se ha dejado de ejecutar el programa, que sería cuando nos piden que introduzcamos una nueva línea, y en ese momento no es posible imprimir por pantalla. Este ha sido uno de los problemas significativos que he tenido y lo he tratado de esta manera.

- IMPLEMENTACIÓN DE JOBS + FG

jobs: Devolverá 1 si se ha intentado ejecutar el mandato "jobs" por pantalla, es decir, al igual que *execcd* o *umask*, se asegura que el nombre del mandato introducido es *jobs.* Devuelve 0 en caso contrario. En caso de que el nombre del mandato a ejecutar fuese *jobs*, se comienza a ejecutar un bucle que guarda en el campo status de cada posición del array de *procesos* el estado actual en el que se encuentra cada proceso. De esta forma, si el proceso está aún ejecutándose se imprime una línea diciendo que se sigue ejecutando. Si el proceso ha terminado, entonces se imprime por pantalla que ha acabado y se elimina del array (se pone el status por defecto y se elimina la línea)

fg: Lo primero que se hace es buscar la posición del array procesos que se ha introducido en último lugar. Para ello se recorre el array desde atrás hacia adelante y se queda con la primera posición válida que encuentre. A continuación se comprueba si el mandato que se quiere ejecutar es *fg* y dependiendo del número de argumentos que se le pase, si no se introducen argumentos utiliza la última posición del array de procesos obtenida tal y como he explicado anteriormente y se espera al final de su ejecución. Si se introduce algún argumento, entonces se comprueba que el argumento sea válido, se transforma a entero y tras obtener dicho proceso del array de procesos, se espera hasta el final de su ejecución. Devuelve 1 si se escribió "fg" como mandato a ejecutar y 0 en caso contrario.

- ESTRUCTURAS DE DATOS ESPECÍFICAS

Las estructuras de datos han sido fundamentales a la hora de resolver ciertos problemas que planteaba la práctica. El array de pipes (array de arrays de dos posiciones) es una estructura a destacar, ya que nos permite comunicar cierto número de procesos. Destacar también que el número de pipes se determina en tiempo de ejecución, una vez sabemos el número de mandatos a ejecutar.

Pero si hay que destacar una estructura de datos utilizada es el array estático de 1024 posiciones llamado *procesos*. Esta estructura de datos es clave para el almacenamiento de los procesos que se ponen en segundo plano (background). Se trata de un array que en cada posición almacena un dato de tipo *background*, que es una estructura interna del programa que se encarga de almacenar el pid (entero) clave de un mandato o secuencia de mandatos separados por pipes. El pid clave se trata de el pid del proceso que ejecuta el último mandato (si es solo un mandato, pues almacena el pid del proceso que lo ejecuta). Esto nos servirá para comprobar si un mandato en background ha terminado de ejecutarse o no, además de identificar a cada proceso en background. En segundo lugar tendremos un puntero a un caracter (o string) que servirá para guardar el nombre completo de la linea que se ha introducido por teclado. En tercer lugar, se declara una variable de tipo entero que servirá para almacenar el estado de ese proceso o conjunto de procesos en background (si se está ejecutando o si ha acabado). En resumidas cuentas, el array *procesos* almacenará de cada línea introducida para ejecutar en background: un pid clave, la línea completa introducida y su estado.

- REDIRECCIONES

He creído conveniente crear una sección dedicada a las redirecciones, ya que tienen mucha relevancia en la práctica y así puedo explicar con más detalle la función *redirect* que he implementado. Dicha función recibe dos parámetros: *redirect(que queremos redirigir, donde queremos redirigir).*

Si lo que se quiere es redirigir la entrada, entonces abrimos un descriptor con permisos de lectura, pero si se quiere redirigir la salida o la salida de error, ambas se tratan de la misma manera ya que lo único que cambia es la opción elegida (STDOUT_FILENO si queremos redirigir salida estándar o STDERR_FILENO si se quiere redirigir salida de error estándar) y se abriría el descriptor con permisos de escritura (o se crearía desde cero en caso de que no exista). Nótese también que se comprueba que la redirección no sea nula, en cuyo caso no se hará ninguna redirección.

Por último, hay que recalcar que el valor de retorno es un entero, que nos servirá para saber si ha habido errores de redirección de entrada (de salida no debería haber errores porque el fichero se crea de nuevo, pero también sirve para conocer si ha habido errores para error y salida estándar). En caso de que haya ocurrido algún error, la función devolverá el valor de la ejecución de *dup2* (-1 en caso de error).

- MANDATOS EXTERNOS

En esta sección se explicará la implementación de mandatos externos. En concreto, nos centraremos en *cd* y *umask.*

cd: Implementado en la función *execcd.* Devuelve 1 si el primer argumento del mandato a analizar es *cd,* devuelve 0 en caso contrario. También se comprueba que la línea a ejecutar no use pipes, si utilizase pipes, se imprime un mensaje de error diciendo que no se puede ejecutar el mandato. Si no usa pipes y el mandato a analizar es *cd* entonces comienza la ejecución en sí.

Se comprueba si el número de argumentos introducidos a *cd* es 0 o 1, si fuese mayor se imprimiría un mensaje de error. Si es 0 entonces se toma como ruta la variable HOME. Si es 1 entonces se concatena la ruta actual con la ruta introducida en el argumento 1 precedido por el caracter '/', de esta forma tendremos la ruta a la que queremos cambiar.

Una vez tenemos la ruta a la que vamos a cambiar, ejecutamos *chdir* con esa ruta y si todo ha ido bien, se imprime por pantalla la variable nuevaruta, que guardará el directorio al que se ha cambiado y lo imprimirá por pantalla. Si ha sucedido cualquier problema, se imprime por la salida de error el error correspondiente.

umask: Implementado en la función *execumask.* Al igual que en la función *cd*, se asegura que sea una linea de un solo mandato y que el comando introducido sea el *umask,* si no se imprime un mensaje de error. Si el número de argumentos con el que se ha llamado a la función *execumask* es 0, entonces se llama a otra función auxiliar (*showmask*) que lo que hace es devolver el valor de la máscara actual. Sin embargo, si se llama con 1 argumento a *execumask*, se comprueba que la expresión introducida esté en octal (para ello se recorre posición a posición el argumento con que se llama a umask y se comprueba que sea un número en octal de 4 cifras e inferior a 0777) y si es así, se cambia la máscara. Si no se imprimiría un mensaje de error en función del error correspondiente.

Principales Funciones

	main	Nombre	Tipo	Descripción
Argumentos	void			
Variables Locales	Variable 1	buf	char[1024]	Sirve para guardar la linea introducida por pantalla en cada iteración
Valor Devuelto			int	0 si el programa se ha ejecutado correctamente
Descripción de la Función	Función a través de la cual el programa comienza a ejecutarse. Contiene llamadas a otras funciones en su interior.			

	execnmandatos	Nombre	Tipo	Descripción
Argumentos	No hay			
Variables Locales	Variable 1	npipes	int	Sirve para guardar el numero de pipes necesarios que se corresponde con uno menos que numero de mandatos introducidos.
	Variable 2	nuevodescriptor	int	Sirve para guardar el valor que devuelve la funcion *redirect* y comprobar posteriormente si ha habido algún problema al redirigir la entrada estándar.
	Variable 3	pid	pid_t	Guarda el pid del hijo creado (o vale 0 si es el proceso padre)
Valor Devuelto			int	0 si todo funciona correctamente
Descripción de la Función	Crea los procesos y pipes necesarios, concatenando la salida del anterior con la entrada del siguiente. Para líneas con más de un mandato. Automatiza la ejecución de n mandatos.			

	execmandato	Nombre	Tipo	Descripción
Argumentos	No hay			
Variables Locales	Variable 1	pid	pid_t	Guarda el pid del hijo creado (o vale 0 si es el proceso padre)
	Variable 2	nuevodescriptor	int	Sirve para guardar el valor que devuelve la funcion *redirect* y comprobar posteriormente si ha habido algún problema al redirigir la entrada estándar.
Valor Devuelto			int	0 si todo ha funcionado correctamente.
Descripción de la Función	Crea un proceso y lo ejecuta. Para líneas de un solo mandato. Establece redirecciones de entrada, salida y error.			

	execumask	Nombre	Tipo	Descripción
Argumentos	No hay			
Variables Locales	Variable 1	nuevamascara	mode_t	Variable donde se van guardando los nuevos valores de la máscara que se pondrá por defecto, tras comprobar que cada uno de ellos es válido
	Variable 2	mascara	char *	Apunta en cada iteración a la siguiente posición (siguiente carácter) del argumento 1 que se le pasa a la función.
	Variable 3	c	char	Guarda el carácter que es apuntado por la variable máscara, para realizar operaciones con él y comprobar si es válido.
Valor Devuelto			int	1 si el nombre del mandato es “umask”, 0 en caso contrario

Descripción de la Función	Cambia la máscara de los permisos por defecto, o imprime por pantalla la máscara actual si no se le pasan argumentos.

	execcd	Nombre	Tipo	Descripción
Argumentos	Argumento 1	line	tline	Variable que contiene la línea introducida parseada
Variables Locales	Variable 1	ruta	char * (o char[1024])	Variable que sirve para guardar la ruta a la que se desea cambiar.
	Variable 2	nuevaruta	char[1024]	Una vez se ha cambiado a la ruta deseada y todo ha ido bien, sirve para guardar el valor de la actual ruta (ruta a la que acabamos de cambiar) para imprimirla por pantalla
Valor Devuelto			int	1 si el nombre del mandato es “cd”, 0 en caso contrario.
Descripción de la Función	Cambia de directorio. Funciona tanto con rutas relativas como absolutas.			

	redirect	Nombre	Tipo	Descripción
Argumentos	Argumento 1	option	int	Descriptor de fichero que se quiere redirigir (STDIN_FILENO para entrada estandar, STDOUT_FILENO para salida estándar y STDERR_FILENO para salida de error estándar)
	Argumento 2	redirection	char *	Nombre del fichero o ruta al que se quiere redirigir
Variables Locales	Variable 1	descriptor	int	Guarda el valor del descriptor al abrir el fichero introducido en el argumento redirection

	Variable 2	nuevodescriptor	int	Guarda el valor generado por dup2. De esta forma nos sirve para conocer si ha sucedido algún error al hacer la redirección.
Valor Devuelto			int	Devuelve nuevodescriptor. -1 Si ha habido algún problema durante la redirección u otro valor si no ha habido problemas
Descripción de la Función	Redirige la entrada, salida y salida de error estándar.			

	jobs	Nombre	Tipo	Descripción
Argumentos	No hay			
Variables Locales	Sin variables locales			
Valor Devuelto			int	1 si el mandato a ejecutar es “jobs”. 0 en caso contrario
Descripción de la Función	Muestra la lista de procesos en background y su estado actual (ejecutándose o terminado).			

	foreground	Nombre	Tipo	Descripción
Argumentos	No hay			
Variables Locales	Variable 1	ultimo	int	Guarda el numero de la última posición en la que se ha introducido un valor en el array de procesos en background.
	Variable 2	posicion	char *	Apunta a la posición del primer argumento introducido por el usuario para transformarlo a entero (guardado en la variable a)

	Variable 3	a	int	Guarda el valor de la posición del array de procesos que se desea traer a foreground
Valor Devuelto			int	1 si el mandato a ejecutar es “fg”. 0 en caso contrario
Descripción de la Función	Trae un proceso a foreground.			

Casos de Prueba

A continuación se muestran una serie de casos de prueba, usados para probar la funcionalidad de la MiniShell y de cada una de las funciones implementadas. Simplemente añadir que muchos de estos casos de prueba han sido repetidos probando diferentes funciones, pero no es posible añadirlos todos por simplicidad y para no extender mucho esta sección. A continuación se muestran los casos de prueba más interesantes o relevantes.

Función	Detalles de la prueba	Salida esperada	Salida	Conclusión y resultado de la prueba
execmandato	Llamada a mandato incorrecto	Error al ejecutar el mandato	Error al ejecutar el mandato	Sí funciona
execmandato, redirect	Llamada a mandato con redirecciones de entrada invalida	Error al ejecutar el mandato con mensaje de error de redirección	Error al ejecutar el mandato con mensaje de error de redirección	Sí funciona
execnmandatos	Llamada a mandatos incorrectos	Error o conjunto de errores al ejecutar el/los mandatos	Error o conjunto de errores al ejecutar el/los mandatos	Sí funciona
execnmandatos, redirect	Llamada a mandatos con redirección de entrada invalida	Error al ejecutar el mandato con mensaje de error de redirección	Error al ejecutar el mandato con mensaje de error de redirección	Sí funciona
execcd	Cambiar a un directorio inexistente	Error al cambiar de directorio	Error al cambiar de directorio	Sí funciona
execcd	Introducir más argumentos de los	Mensaje de error. Número de	Mensaje de error. Número de	Sí funciona

	permitidos	argumentos no válido	argumentos no válido	
jobs	Llamada a jobs sin procesos en la lista de background	No hay salida	No hay salida	Sí funciona
foreground	Llamada a fg sin procesos en la lista de background	Imprime mensaje de error	Imprime mensaje de error	Sí funciona
exit	Llamada a exit con varios argumentos	Sale del programa, dan igual los argumentos	Sale del programa, dan igual los argumentos	Sí funciona
background	Ejecucion de find en el directorio raiz en background. Despues pulsar Ctrl+C	No se detiene la ejecucion	No se detiene la ejecucion	Sí funciona
background	Llamada a un mandato incorrecto	Se añade a la lista de procesos, pero se imprime un mensaje de error y se termina el proceso	Se añade a la lista de procesos, pero se imprime un mensaje de error y se termina el proceso	Sí funciona
umask	Cambiar permisos a un número inválido	Mensaje de error	Mensaje de error	Sí funciona
cd, umask, jobs, foreground	Concatenar con pipes los mandatos	Mensajes de error	Mensajes de error	Sí funciona
redirect	Redirigir salida y salida de error a un fichero que no existe	Creación del nuevo fichero e impresión del error	Creación del nuevo fichero e impresión del error	Sí funciona
foreground, background	Ejecutar varios mandatos en background, y traer uno a foreground	Ejecución en foreground del proceso en foreground e impresión por pantalla de los procesos que han terminado	Ejecución en foreground del proceso en foreground e impresión por pantalla de los procesos que han terminado	Sí funciona
exit	Salir del programa con procesos en background	Salida exitosa del programa	Salida exitosa del programa	Sí funciona

Comentarios Personales

- PROBLEMAS ENCONTRADOS

El problema principal que he encontrado con respecto a esta práctica ha sido el tiempo que conlleva. Dado que nunca antes hemos trabajado con procesos, esta nueva forma de programar, junto con el extenso temario de la asignatura hace que manejar todos los conceptos en profundidad y correctamente lleve su tiempo de asimilar. Por ello en propuestas de mejora he escrito alguna sugerencia en relación con este problema.

Al ser una práctica que requiere cierto tiempo, a lo largo de su transcurso han ido apareciendo problemas que pueden ser destacables. Un problema encontrado ha sido, en la implementación del tratamiento de señales, al tratar SIGINT (Ctrl + C) mientras se lee una nueva línea, no se volvía a mostrar el prompt, puesto que después de llamar a la señal el programa vuelve al punto donde dejó de ejecutarse. Esto hacía que no se pudiese imprimir por pantalla la nueva línea con el prompt. Una vez detectada la raíz de este problema (radica en que el fgets, una vez que pide que se introduzca algo por pantalla, no se puede detener a no ser que se ejecute Ctrl + C o se introduzca la nueva línea) se ha llegado a la conclusión de que no había forma de imprimir una línea con el nuevo prompt sin salir del programa, y se ha optado por implementar una alternativa, que es desactivar la señal SIGINT durante la lectura de la nueva línea.

Otros problemas encontrados han sido en relación con el planteamiento de mandar información entre procesos. Planificar cómo se deben comunicar los procesos mediante el array de pipes ha sido sin duda uno de los retos de esta práctica. Sin embargo han sido solventados exitosamente.

Por último, la implementación del background ha supuesto todo un reto, pero ha sido muy útil, en especial para manejar nuevas estructuras de datos y comprender mejor el proceso de espera de un hijo con WNOHANG.

- CRÍTICAS CONSTRUCTIVAS

La implementación de esta práctica ha servido para aprender con detalle todo lo relativo a procesos pesados, uso de tuberías, uso de los descriptores de ficheros, entre otras cosas. Aunque al principio, al leer el enunciado parece un poco abrumador, debido a la gran cantidad de detalles que hay que tener en cuenta. Por ello quizás sería buena idea dar algún ejemplo de cómo se espera que funcione algún mandato (por ejemplo, al principio yo no entendía qué se quería decir con la frase sobre el mandato umask que dice lo siguiente: “los permisos por defecto que se establezcan serán el resultado de la resta bitwise del número octal pasado como argumento al mandato”). Algún ejemplo de cómo se desea que sea la salida, o del propio funcionamiento del programa puede ser de gran utilidad y siempre es bienvenido por los estudiantes.

- PROPUESTAS DE MEJORA

Creo que esta es una muy buena práctica, que ayuda a reforzar gran parte de los contenidos vistos en la asignatura. Apenas se me ocurren propuestas de mejora con respecto a esta práctica, sin embargo voy a mencionar una propuesta que está relacionada con el problema principal que he encontrado durante su implementación: el tiempo. Como ya he mencionado antes, asimilar todos los conceptos lleva tiempo. Por ello una buena sugerencia sería dejar algo de tiempo más para llevar a cabo todas las funcionalidades de la práctica. Dejar que la fecha de entrega de la práctica sea más tarde nos puede ayudar a organizarnos mejor como individuos, ya que otras entregas del resto de asignaturas nos quitan también tiempo.

Sin embargo, en otros aspectos no tengo nada que objetar. Me parece una práctica completa, bien organizada y que ayuda a comprender muchos de los conceptos estudiados.

- EVALUACIÓN DEL TIEMPO DEDICADO

Creo que el tiempo dedicado en mi caso ha sido bueno. Siento que dependiendo del día, hay días que puedo ser muy productivo y otros en los que estoy atascado en un error o en un problema mucho tiempo y no avanzo nada.

A rasgos generales, creo que el esfuerzo empleado ha sido notablemente mayor que con el de la práctica 1, pero también ha sido un tiempo y esfuerzo bien invertido, puesto que me ha ayudado a conocer muchos de los conceptos que se habían visto teóricamente en clase y que ponerlos en práctica ha sido algo muy satisfactorio.

Como comentaba en el punto anterior, sería buena idea ampliar el plazo de entrega de la práctica, puesto que manejar bien los procesos con fork en un breve período de tiempo es complicado, debido a que es una forma de programación distinta a todo lo que habíamos visto hasta ahora y un simple error puede hacer que el programa funcione de manera muy distinta a la que se desea. Por no mencionar, que como los errores en C son muy poco descriptivos, es difícil saber dónde hay un error a no ser que conozcas bien tu código.

En general creo que ha sido una muy buena práctica, que te ayuda a conocer mejor muchos conceptos y a manejar los procesos y la comunicación entre ellos. Los profesores han estado siempre dispuestos a ayudar y eso también es digno de destacar.