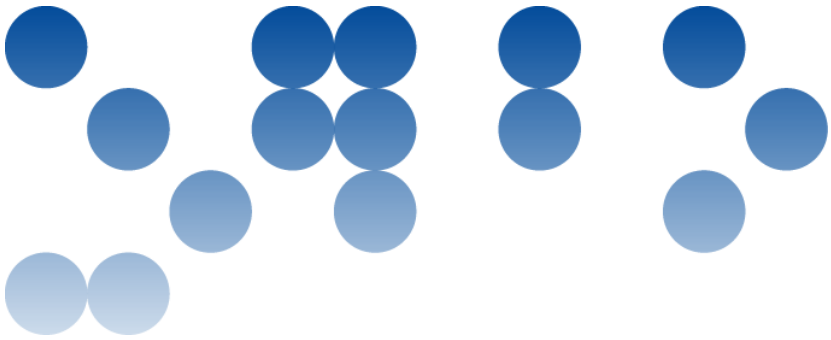


PRA1
Introducción a la programación en Linux

Sistemas Operativos

Programa
2018-01

Estudios de Informática, Multimedia y Telecomunicación

UOC
Universitat Oberta de Catalunya

Presentación

Esta práctica plantea una serie de actividades con el objetivo que el estudiante pueda aplicar sobre un sistema Unix algunos de los conceptos introducidos en los primeros módulos de la asignatura.

El estudiante tendrá que realizar una serie de experimentos y responder las preguntas planteadas. También tendrá que escribir unos pequeños programas en lenguaje C.

La práctica se puede desarrollar sobre cualquier sistema Une (la UOC os facilita la distribución Ubuntu 14.04). Se aconseja que mientras realizáis los experimentos no haya otros usuarios trabajando al sistema porque el resultado de algunos experimentos puede depender de la carga del sistema.

Cada pregunta indica una posible temporización para poder acabar la práctica antes de la fecha tope y el peso de la pregunta a la evaluación final de la práctica.

El peso de esta práctica sobre la nota final de prácticas es del 40%.

Competencias

Transversales:

- Capacidad para adaptarse a las tecnologías y a los futuros entornos actualizando las competencias profesionales

Especificas:

- Capacidad para analizar un problema en el nivel de abstracción adecuado a cada situación y aplicar las habilidades y conocimientos adquiridos para abordarlo y resolverlo.
- Capacidad para diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración y reutilización.

Competencias propias:

- El uso y aplicación de las TIC en el ámbito académico y profesional.

Objetivos

Esta práctica tiene como objetivo que el alumno aplique los conceptos introducidos en los 4 primeros módulos de la asignatura sobre un sistema Unix y que se introduzca de forma progresiva en la programación, depuración y ejecución de aplicaciones en los sistemas Unix.

Restricciones en el uso de funciones de C

Para realizar la práctica debe utilizar las llamadas al sistema de Unix. **No se podrán utilizar funciones de C de entrada / salida** (getc, scanf, printf, ...), en su lugar se utilizarán las llamadas a sistema read y write. Sí que puede usar funciones de C para el formateo de cadenas (sprintf, sscanf, ...).

Enunciado

Os facilitamos el fichero pr1so.zip con una serie de archivos que son útiles para realizar la práctica. Descompactarlo ejecutando el comando unzip pr1so.zip.Per compilar un fichero, por ejemplo prog.c, hay que ejecutar el comando gcc -o progprog.c y para ejecutar el programa hay que hacer ./prog

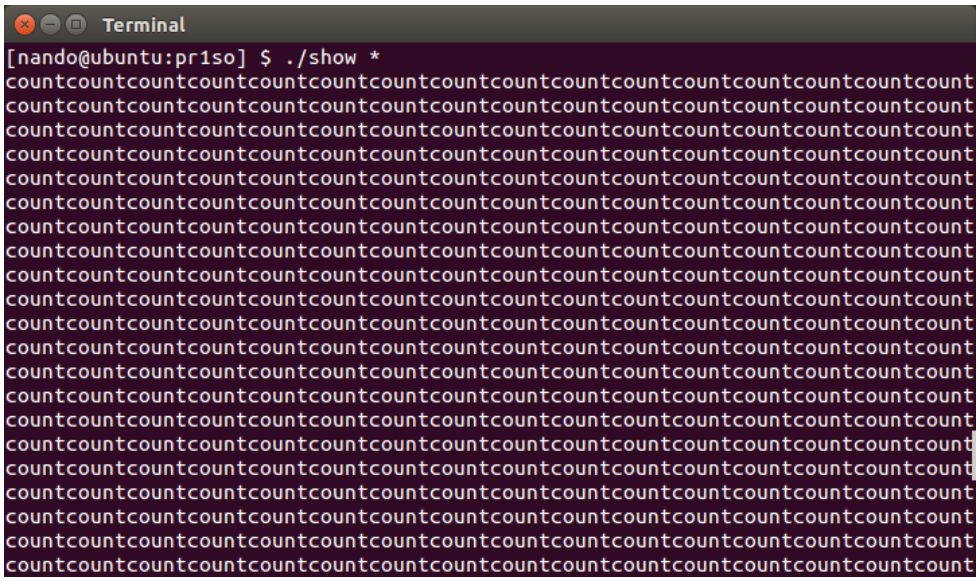
Como referencia, a cada pregunta os aconsejamos una posible temporización para poder terminar la practica antes de la fecha límite. También se indica el peso de cada pregunta a la evaluación final de la práctica.

1: Módulo 2 [Del 5 al 13 de Octubre] (30%, todos los apartados tienen el mismo peso)

Os facilitamos el programa show.c que imprime 1000 veces por pantalla el carácter que se le pasa:

Figura 1. Ejemplo de ejecución programa show.

1.1 En la ejecución del ejemplo anterior (figura.1) le pasamos el carácter + al programa show y nos lo muestra 1000 veces por pantalla. Sin embargo, si le pasamos un asterisco en el programa obtenemos el siguiente resultado:



Teneda en cuenta que en su sistema la salida puede variar de la mostrada en el ejemplo. Esta salida asume que ejecute el programa desde el directorio que contenga descomprimido el f itxer pr1so.zip.

¿Porque se obtiene este resultado?

Porque el asterisco es un meta carácter que bash se entiende como un patrón de búsqueda en los archivos del directorio actual. Al final, pasa al programa show todos los archivos/directorios del directorio actual y el programamuestra el primero de ellos 1000 veces por pantalla.

Como habría que invocar al programa para que se obtenga el resultado esperado (impresión de 1000 asteriscos por pantalla)?

Habría que invocar el programa con el * entre comillas, indicando que es un carácter:

> ./show '*'

1.2 Ejecutad y justificar el resultado obtenido al ejecutar cada una de las siguientes órdenes:

- show +; show -; show /

  Esta orden falla porque no se especifica la ruta del programa show, por lo tanto, lo busca en los directorios del $PATH y al no encontrar el programa genera un error.

- ./show +; ./show -; ./show /

  El carácter ';' permite realizar la ejecución de forma secuencial de varios programas. Por tanto, esta orden ejecuta correctamente cada uno de las tres invocaciones de show, con diferentes caracteres, de forma secuencial.

- ./show + || ./show - || ./show /

  La expresión || se utiliza en bash-script como operador condicional OR. Al utilizarlo para concatenar comandos significa que la siguiente orden de la lista se ejecutará sólo si la ordenanterior falla (devuelve falso).

  La ejecución del primer *show* hace que se muestre por pantalla 1000 veces el carácter + y que el programa vuelva el código de error de 0 (exit (0)), indicando que se ha ejecutadocorrectamente. Esto hace que el resto de órdenes no se ejecuten.

- ./show + && ./show - && ./show /

  El símbolo && utiliza en bash-script como operador condicional AND, y al contrario que con el comando anterior, sólo permite la ejecución del siguiente comando si las órdenesprevias se hanejecutado correctamente.

  La ejecución de esta orden, invoca secuencialmente cada uno de los tres comandos (ya que todos terminan correctamente devolviendo un 0) y, por tanto, se mostrará por pantalla primero 1000 veces el carácter +, después 1000 veces el carácter -, y finalmente el carácter / 1000 veces más.

- ./show + & ./show - & ./show /

  El carácter & permite la ejecución concurrente de una orden. En este caso, lo que está indicando que se ejecuten de forma concurrente las tres órdenes show. Es por ello, que la salida de cada uno de los comandos se muestra mezclada entre sí.

1.3 Ahora utilizaremos el programa show2.c, que es una variante del programa anterior y que imprime un carácter de forma indefinida.

Ejecutad el siguiente comando (mini-script) con el programa show2.

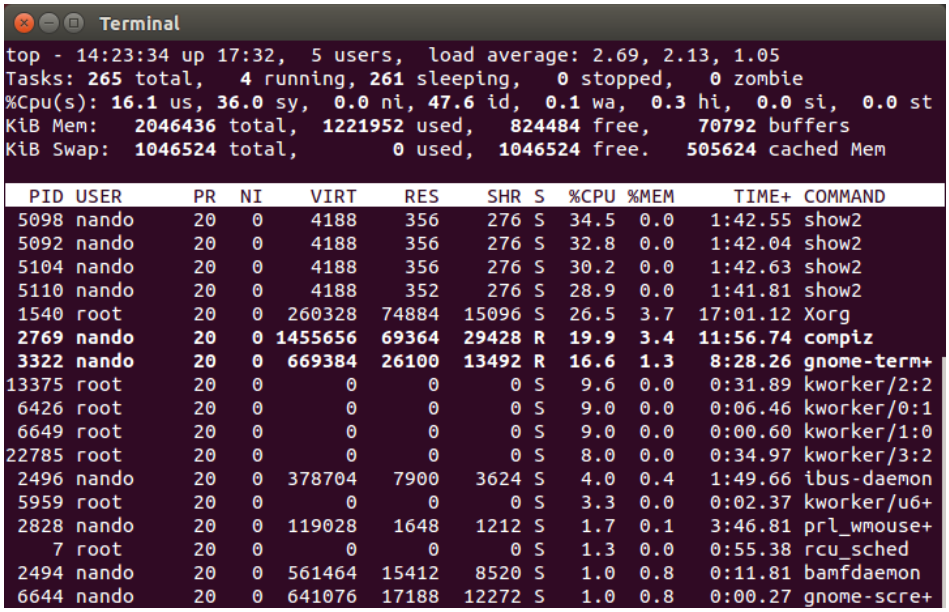
```
> for ((i=0; i<`cat /proc/cpuinfo | grep "cpu cores" | uniq
  | cut -d: -f2`; i++)) do  ./show2 $i &done
```

a) Indicar que hace este comando y el resultado que mostrará por pantalla.

Ejecuta tantos programas show concurrentes como cpus disponga el procesador donde estamos ejecutando la orden. El primer programa show mostrará un 0 por pantalla, el siguiente un 1 y así sucesivamente.

El mini-script implementa un for que va desde 0 hasta el número de procesadores de la máquina (obtenidos consultando el archivo cpuinfo del directorio /proc). En cada iteración de este bucle, ejecuta el programa show de forma concurrente, pasándole el índice del bucle como carácter a mostrar por pantalla.

b) Abre otra consola y utiliza el comando top para monitorizar la ejecución de la orden anterior. Adjuntar una captura de pantalla del top y justificad los resultados obtenidos.



Podemos comprobar que se ejecutan 4 procesos con el programa show2, ya que mi procesador dispone de 4 núcleos.

También, podemos ver que estos procesos no están consumiendo toda CPU del sistema, ya que cada uno de ellos apenas consume el 35% de CPU y que a nivel global casi el 48% de los recursos de cómputo están ociosos. También como era de esperar, estos programas consumen muy poca memoria.

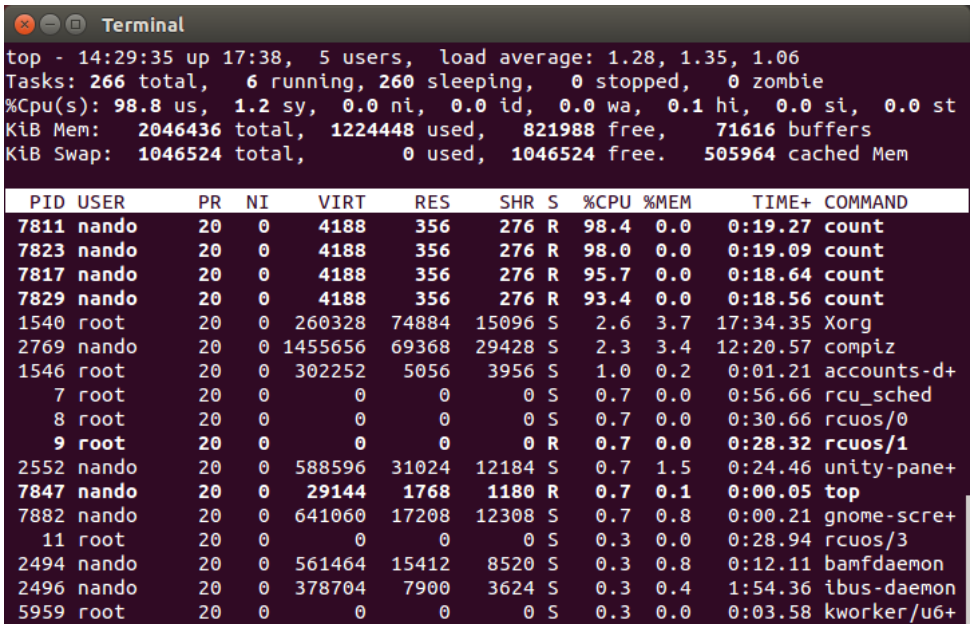
Podéis matar todos los procesos arrancados con el comando anterior utilizando el comando: **killall -9 show2**

1.4 Comparar los resultados obtenidos de la ejecución del comando anterior con los obtenidos por la a siguiente orden:

```
> for ((i=0; i<`cat /proc/cpuinfo | grep "cpu cores" | uniq
  | cut -d: -f2`; i++)) do  ./count $i &done
```

Ahora se ejecuta el programa *count.c* que ejecuta el incremento de una variable dentro de un bucle infinito.

Adjuntar una captura de pantalla del top durante la ejecución del comando.



A que se debe el alto porcentaje de utilización de la CPU de los procesos *count*, tanto a nivel global como individual?

El programa *count* ejecuta un bucle infinito en el que cada iteración incrementa una variable. Esto implica que el proceso está todo el tiempo ejecutando instrucciones (x++) ynunca sebloquea para hacer E/S, de esta forma nunca deja el estado de Run. Esto implica que el proceso intenta utilizar toda la cpu disponible. Además, al haber tantos procesos *count*como núcleos tenga el procesador, cada uno de ellos puede utilizar uno de los núcleos casi al 100% y con ello todo el sistema se encuentra saturado, ocupado al 100%.

los procesos show2 presentan el mismo comportamiento? A que se debe esta diferencia de comportamiento?

No, los procesos show2, aunque también ejecutan un bucle infinito, no consumen el 100% de la CPU ni a nivel individual ni global. En concreto, el consumo de CPU de estosprocesos no supera el 35% individualmente ni el 54% de globalmente.

Esto se debe a que dentro de su bucle infinito, los procesos show2 realizan E/S por pantalla (write) lo que obliga a que estos procesos pasen al estado de Wait, liberando la CPU. Hasta que la E/S no se complete, estos procesos no pueden volver a ejecutarse. Esto implica que durante este periodo de tiempo la CPU está disponible para ser utilizada por otros procesos,o quedar ociosa (i.e. baja la ocupación del sistema).

2 Módulo 3[Del 14 al 24 de octubre] (40% = 10% + 15% + 15%)

En este apartado se utilizará el programa *fib.c* que calcula el enésimo número de la serie de Fibonacci de forma recursiva.

2.1 Primero depuraremos algunos errores de memoria que tiene el programa original. Probar las siguientes invocaciones del programa *fib*, verificar si se ejecutan correctamente o si fallan (normalmente generando una excepción de "segmentationfault "). En caso de que fallen, indicad cuál es el error que generan e identificar la causa de dicho error (justificad las respuestas):

- ./fib

  Genera un error de segmentación ("Segmentationfault") debido a que el programa supone que se le pasará un parámetro. Si no le pasa ninguno, entonces cuando intenta hacer la conversión del parámetro a un entero se produce la excepción.

- ./fib 8

  Se ejecuta correctamente, devolviendo el resultado correcto (21).

- ./fib -1

  En este caso se genera también una excepción "Segmentationfault". Sin embargo, esta vez el error se produce debido a que se genera una recursividad demasiado grande y esto provoca que se agote la pila del proceso y la consiguiente excepción.

- for ((n = 1; n <10000000; n = n * 10)) do ./fib $n; done

  En este caso se podría generar una excepción “Segmentationfault” en las últimas ejecuciones del bucle. Este error se debe a que cuando el resultado tiene más de 2 dígitos se está desbordando la capacidad del buffer str (que tiene 30 bytes/caracteres). Por lo tanto, al formatear la cadena de salida se está escribiendo más allá de la memoria asignada a str y en función de lo crítica que sea la información que se borra se puede producir la excepción.

Observaciones:

- Para detectar y corregir los errores, podéis utilizar el depurador gdb, con los archivos de core (volcado de la memoria del proceso) que genera un programa cuando se finaliza por una excepción.

  Tal como se puede ver en la siguiente captura de pantalla, para depurar estos errores, primero se incluirá información de depuración en el ejecutable (opción -g del compilador), se debe activar la generación de los ficheros core ("ulimit -c unlimited") y posteriormente ejecutar el programa que falla.

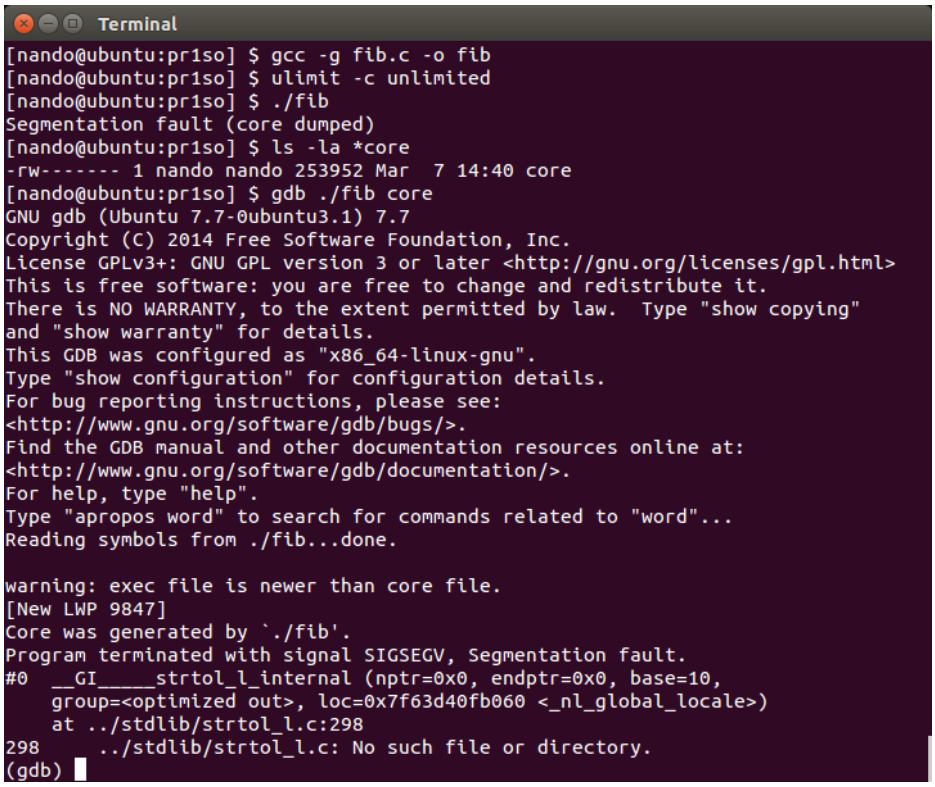


Figura 2. Procedimiento para la depuración de excepciones.

Una vez producido el error, se invoca al depurador (gdb), indicando el nombre del programa que se quiere depurar y el fichero de core generado ("gdb ./fibcore"). Utilizando el comandobt del depurador, podremos obtener la pila de llamadas y ver cuál es la línea de código de nuestro programa que ha generado la excepción.

2.2 Corregir los problemas de memoria del programa /fib.c. Adjuntad una copia del programa modificado indicando en los comentarios del mismo el código añadido para corregir los errores. El programa obtenido no

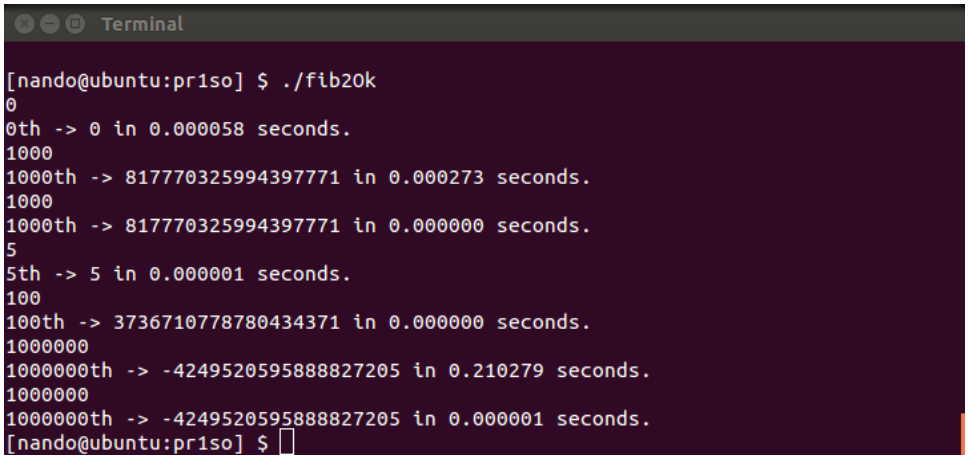
debería fallar, independientemente de la longitud del número de la serie que se quiera obtener.

Para corregir el último error hay dos alternativas:

- Limitar la recursividad o limitar el número máximo de la serie de fibonacci que se puede calcular. Esta versión la tenéis en el fichero fibOk.c
- Calcular la serie de fibonacci de forma iterativa. Esta versión la tenéis en el fichero fibOk_b.c

2.3 Os adjuntamos una segunda versión iterativa, fib2.c, para calcular los números de la serie de Fibonacci. Esta versión pide al usuario que introduzca el número de la serie que quiere calcular y le devuelve el resultado, junto con el tiempo que ha tardado en hacerlo.

Modificar este programa para que a medida que se van calculando los elementos intermedios de la serie de Fibonacci, estos se guarden en un vector. Posteriormente se podrá utilizar estos resultados pre-calculados para reducir el tiempo de cálculo de los elementos de la serie. Para evitar la necesidad de formatear los números para mostrarlos por pantalla, los elementos de la serie se guardarán en formato de cadena.



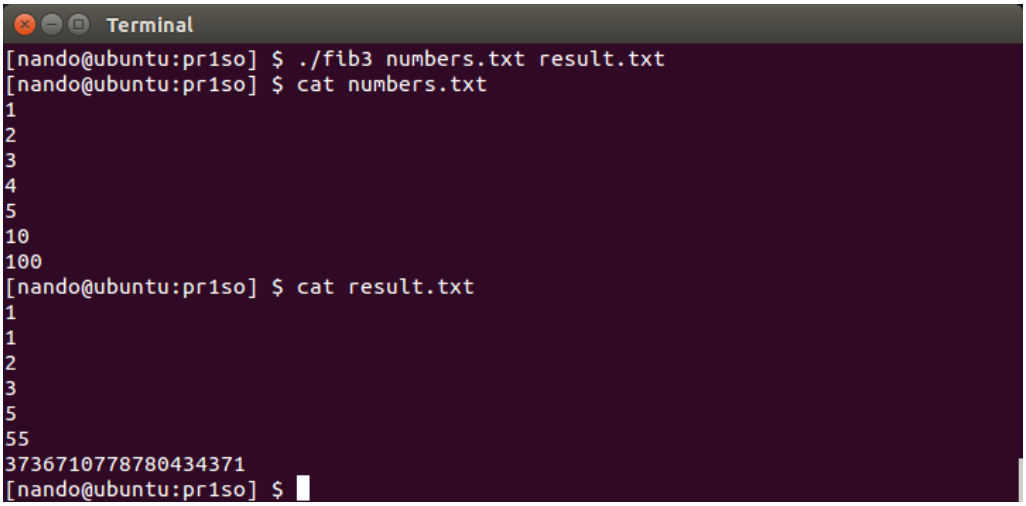
Para implementar la gestión de memoria dinámica podéis utilizar las funciones *malloc, realloc* y *free* de la biblioteca malloc de C. Se valorará que la resolución del problema no solicite más memoria de la estrictamente necesaria.

La solución a este aparte la tiene en el fichero fib2Ok.c

3 **Módulo 4**[Del 25 al 31 de octubre] (30% = 15% + 15%)

En este apartado vamos a servir la última versión del programa fib3.c, para el cálculo de los números de la serie de Fibonacci. En esta versión el usuario pasa dos ficheros por parámetro: el primero es un fichero de entrada que contiene los números de la serie que se quiere calcular y en el segundo un fichero de salida donde se grabará el resultado.

El comando que hay que utilizar para ejecutar esta versión es el siguiente:



3.1 Estudiar el código del programa y explicar cada una de las operaciones utilizadas para hacer la lectura y la escritura a archivo. Explicar las funciones/llamadas al sistema utilizadas, los parámetros que se pasan y los valores de retorno.

Podéis utilizar las páginas del manual para obtener información sobre estas operaciones. Por ejemplo:>man 2 open

- fdin = open (filein, O_RDONLY);

  Abre el archivo de entrada (filein) en modo sólo lectura y asigna el descriptor del fichero a la variable fdin. si se produce un error fdin tendrá un valor negativo.

- fdout = open (fileout, O_WRONLY | O_CREAT | O_TRUNC, 0660);

  Abre el archivo de salida (fileout). Se especifica el modo de acceso (O_WRONLY, sólo escritura), los flags de creación (O_CREAT, si no existe el fichero se creará) y de borrado (O_TRUNC, si el archivo ya existe se borrará) y los permisos de acceso (0660, lectura y escritura para el propietario y el grupo del propietario). Si la apertura tiene éxito el descriptor de archivo se asigna a la variable fdout. Si se produce un error fdout tendrá un valor negativo.

- (ret=read(fd,&number_str[x++],1))>0

La llamada al sistema read permite leer de un descriptor de archivo. En este caso, se intenta leer un número carácter a carácter, para posteriormente convertirlo a entero. En el primer parámetro especifica el descriptor de archivo a leer (fd). En el segundo parámetro el buffer o la posición de memoria donde se guardará la información leída (&number_str[x++]). Finalmente, el tercer parámetro indica cuántos bytes se quiere leer (en este caso sólo uno).

La llamada al sistema devuelve el número de bytes que se han leído, 0 si se ha llegado al final del archivo y -1 si se ha producido un error.

- if (write (fdout, str, strlen (str)) <0)

  La llamada al sistema write permite escribir información en un archivo. En este caso, se escribe una cadena en el fichero asociado con el descriptor fdout. La información a escribir se especifica mediante su dirección en el segundo parámetro (str) y su longitud en el tercer parámetro (strlen(str), longitud de la cadena).

  La llamada al sistema devuelve el número de bytes que se han conseguido escribir en el fichero o -1 si se produce un error.

- close (fdin);

  Cierra el archivo de entrada.

- close (fdout);

  Cierra el archivo de salida.

3.2 Indicar cual sería el resultado de ejecutar los siguientes comandos y como se tendría que modificar el programa fib3 para soportarlos.Se os adjunta un fichero (comandos_apartado_3.2.txt) con estas ordenes para que no haya problemas de interpretación con los caracteres especiales (pipes, redirecciones).

- ./fib3 < numbers.txt

  Redirige el archivo numbers.txt a la stdin del programa fib3. En este caso, al no especificar ningún archivo, el programa debería leer los números de la entrada estándar (stdin) y generar el resultado por la salida estándar (stdout).

- ./fib3 numbers.txt > result.txt

  Se pasa al programa el fichero de entrada numbers.txt desde donde se leerán los números a calcular y se redirige la stdout al archivo result.txt. Por lo tanto, se espera que el programa fib3 sea capaz de leer los datos de entrada del archivo numbers.txt y generar los

resultados en la stdout (que posteriormente el usuario puede redirigir a un archivo)

- ./fib3 numbers.txt 2> error.txt

  Se pasa al programa el fichero de entrada numbers.txt desde donde se leerán los números a calcular, no se especifica un archivo para los resultados de salida y se redirige la salida estándar de error (stderr) al archivo error.txt.

  El programa fib3 debe poder leer los datos de entrada del archivo numbers.txt y generar los resultados por la stdout (que posteriormente el usuario puede redirigir a un archivo) y generar todos los mensajes de error por la stderr (descriptor 2).

- ./fib3 numbers.txt | sort -nr

  Se pasa al programa el fichero de entrada numbers.txt desde donde se leerán los números a calcular y se redirige la stdout mediante una tubería al comando sort.

  Por lo tanto, el programa leerá los datos de entrada del archivo numbers.txt y generará los resultados en la stdout, que posteriormente el usuario puede redirigir mediante una pipe hacia otro comando (en este caso sort). Para que esto sea posible, los resultados del programa no deben mezclarse con los errores, los cuales siempre se deberán escribir por la stderr.

Asumir que el archivo numbers.txt contiene los números de la serie de Fibonacci que queremos calcular, que en result.txt obtendremos el resultado (los números de la serie) y que en err.txt se guardarán los errores cometidos.

Con la versión actual del programa fib3, las órdenes anteriores no funcionan. Realizar las modificaciones necesarias para que estas órdenes funcionen.

La solución a este aparte la tiene en el fichero fib3_2Ok.c

Recursos

Recursos Básicos

- Documento "Introducción a la programación de Unix" (disponible en el campus virtual).
- Documento "Intérprete de comandos UNIX" (disponible en el aula) o cualquier otro manual similar.

- Cualquier manual básico de lenguaje C.
- El aula "Laboratorio de Sistemas Operativos" (puede plantear sus dudas relativos al entorno Unix, programación, ...).

Recursos Complementarios

- Manual de llamadas al sistema instalado en cualquier máquina UNIX (comando man).

Criterios de valoración

Cada uno de los apartados de la práctica tiene un peso del 50% sobre la puntuación final. El peso de cada apartado se distribuye de forma equitativa entre todos los subapartados que lo integran.

En la corrección se tendrán en cuenta los siguientes aspectos:

- Las respuestas deberán estar articuladas a partir de los conceptos estudiados en teoría y en la guías de la asignatura.
- Se valorará esencialmente la correcta justificación de las respuestas.
- Se agradecerá la claridad y la capacidad de síntesis en las respuestas.
- La capacidad de análisis de los resultados obtenidos y la metodología seguida para su obtención.
- El código entregado debe poder ejecutarse correctamente en cualquier máquina con Linux. Que el código se ejecute correctamente en su ordenador, no implica necesariamente que sea correcto. Revisar varias veces su solución y no ignorar los warnings que pueda estar generando el compilador.

Formato y fecha de entrega

Se creará un archivo zip que contenga un fichero pdf con la respuesta a todas las preguntas y los archivos .c con el código fuente delos distintos ejercicios.

El nombre del archivo tendrá el siguiente formato:"Apellido1Apellido2PRA1.zip". Los apellidos se escribirán sin acentos. Por ejemplo, el estudiante Marta Vallès y Marfany utilizará el nombre de archivo siguiente: VallesMarfanyPRA1.tar

La fecha límite para la entrega de la práctica es el **miércoles 31 de octubre de 2018.**

Nota: Propiedad intelectual

A menudo es inevitable, al producir una obra multimedia, hacer uso de recursos creados por terceras personas. Es por lo tanto comprensible hacerlo en el marco de una práctica de los estudios del Grado Multimedia, siempre y esto se documente claramente y no suponga plagio en la práctica.

Por lo tanto, al presentar una práctica que haga uso de recursos ajenos, se tiene que presentar junto con ella un documento en que se detallen todos ellos, especificando el nombre de cada recurso, su autor, el lugar donde se obtuvo y su estatus legal: si la obra está protegida por el copyright o se acoge a alguna otra licencia de uso (CreativeCommons, licencia GNU, GPL ...). El estudiante tendrá que asegurarse que la licencia que sea no impide específicamente su uso en el marco de la práctica. En caso de no encontrar la información correspondiente tendrá que asumir que la obra está protegida por el copyright.

Habrán, además, adjuntar los ficheros originales cuando las obras utilizadas sean digitales, y su código fuente si corresponde.

Otro punto a considerar es que cualquier práctica que haga uso de recursos protegidos por el copyright no podrá en ningún caso publicarse en Mosaic, la revista del Graduado en Multimedia a la UOC, a no ser que los propietarios de los derechos intelectuales den su autorización explícita.