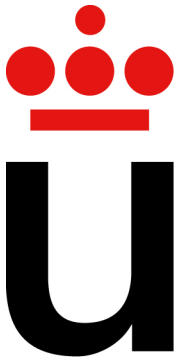

Universidad
Rey Juan Carlos

Sistemas Operativos

[PRÁCTICA 1 – PROGRAMACIÓN DE UNA LIBRERÍA EN C]

DANIEL SANTOS LÓPEZ / DIEGO SÁNCHEZ RINCÓN

TABLA DE CONTENIDO

Autores

Grado en Ingeniería Software – 2024/25

Nombre	GitHub	LinkedIn
Daniel Santos López	danisntoss	Daniel Santos López
Diego Sánchez Rincón	CuB1z	Diego Sánchez Rincón

Descripción del Código

Diseño del Código

Librería.h

Fichero que contiene las cabeceras de las funciones en la librería.

Librería.c

Fichero que contiene la implementación de las distintas funciones en la librería.

- Head (int N): int
  - Emula el mandato `head` de la bash, devuelve por salida estándar las N primeras líneas que recibe por entrada estándar.
  - Definimos las variables
    - Len: de tipo size_t. Guarda la longitud de una línea (tomamos 1024 bytes por defecto).
    - Buffer: de tipo string (array de caracteres).
    - i: de tipo entero.
  - Reservamos memoria de forma dinámica para el buffer mediante la función malloc().
  - Iteramos tantas veces como indique el parámetro N.
    - Leemos una línea de la entrada estándar mediante la función getline() y se almacenan los bytes leídos.
    - Controlamos el error comprobando si el número de bytes leídos es -1.
    - Imprimimos la línea por salida estándar.
  - Liberamos la memoria reservada previamente.

- Tail (int N): int
  - Emula el mandato `tail` de la bash, devuelve por salida estándar las N últimas líneas que recibe por entrada estándar.
  - Definimos las variables:
    - Len: de tipo size_t. Guarda la longitud de una línea (tomamos 1024 bytes por defecto).
    - Buffer: de tipo String (array de caracteres).
    - i: de tipo entero.
    - Data: array de strings.
    - Current: de tipo entero, inicializado a 0 (porque la primera posición es el 0).
  - Reservamos memoria para almacenar los datos. Primero reservamos espacio para N punteros a Strings. Luego iteramos sobre N para reservar memoria de forma dinámica para cada una de las líneas que vamos a almacenar mediante la función malloc().
  - Reservamos memoria de forma dinámica para el buffer mediante la función malloc().
  - Leemos líneas de entrada estándar hasta el fin del fichero:
    - Copiamos la línea leída en la memoria, hasta llenar el array.
    - Una vez lleno, sobrescribimos la posición correspondiente ayudándonos con la variable current.
  - Iteramos sobre el array data (que contiene la salida resultante):
    - Imprimimos por salida estándar, empezando por la posición current, el valor más antiguo leído.
    - Liberamos la memoria que albergaba la línea impresa.
    - Incrementamos la variable current.
    - Si el contador ha llegado a N, reiniciamos current a 0 (para mantener el orden correspondiente).
  - Liberamos la memoria de las variables data y buffer.

- Longlines (int N): int
  - Imprime las N líneas más largas recibidas por entrada estándar
  - Definimos las variables:
    - Len: de tipo size_t. Guarda la longitud de una línea (tomamos 1024 bytes por defecto).
    - Buffer: de tipo String (array de caracteres).
    - i: de tipo entero.
    - Data: array de strings.
    - Count: de tipo entero, inicializado a 0 (porque la primera posición es el 0).
    - WorstLine: variable auxiliar de tipo entero que guarda la longitud de la línea más corta en Data
    - WorstLineIndex: variable auxiliar de tipo entero que guarda la posición de la línea más corta en Data.
    - Lengths: array de enteros que guarda las longitudes de las líneas en Data.
  - Reservamos memoria de forma dinámica para las longitudes de N posiciones mediante la función malloc().
  - Reservamos memoria para almacenar los datos. Primero reservamos espacio para N punteros a Strings. Luego iteramos sobre N para reservar memoria de forma dinámica para cada una de las líneas que vamos a almacenar mediante la función malloc().
  - Reservamos memoria de forma dinámica para el buffer mediante la función malloc().
  - Leemos líneas de entrada estándar hasta el fin del fichero:
    - Asignamos a lineLength la longitud de la línea que acabamos de leer.
    - Si no hemos llenado el array data:
      - Copiamos la línea leída a la posición correspondiente (esto lo lleva la variable count)
      - Actualizamos en el array lengths la longitud de la línea que acabamos de almacenar.
      - Incrementamos el contador.
    - Si ya está lleno el array data:
      - Inicializamos la peor línea como la primera.
      - Iteramos sobre el resto de las líneas para comprobar cuál es realmente la peor, actualizando worstLine y worstLineIndex cada vez que encontramos una peor.
      - Comprobamos si la longitud de la línea que acabamos de leer es mayor que la peor de nuestro array Data. Si es así, reemplazamos la peor línea con la nueva.
  - Ordenamos el array Data resultante (una vez leídas todas las líneas) mediante Quicksort utilizando la función compareStringsByLength.
  - Iteramos sobre el array Data:
    - Imprimimos por salida estándar, empezando por la primera posición, que contiene la línea más larga.
    - Liberamos la memoria que albergaba la línea impresa.
  - Liberamos la memoria de las variables data y buffer.

- CompareStringsByLength (const void * a, const void * b): int
  - Función auxiliar que utiliza el Quicksort para comparar 2 variables.
  - Devuelve 0 si la longitud de los strings es igual.
  - Devuelve un número positivo si la longitud de a es mayor a la de b.
  - Devuelve un número negativo en caso contrario.

Test.c

- Constantes:
  - Flags: array de strings que contiene los argumentos permitidos, estos son “-head”, ”-tail“, ”-longlines”.
  - Usage: string que contiene las instrucciones de uso del test.
  - Default_Lines: entero que contiene el número de líneas por defecto (10)
- Main (int argc, char* argv[]):
  - Es el programa principal del test.
  - Variables:
    - N: entero que guarda el número de líneas que especifica el usuario.
    - Selection: string que guarda el “mandato” que introduce el usuario por entrada estándar.
  - Control de argumentos:
    - Si el número de argumentos es igual a 3:
      - Actualizamos el valor de selection con argv[1]
      - Actualizamos el valor de N con la conversión de argv[2] a entero
    - Si el número de argumentos es igual a 2:
      - Actualizamos el valor de selection con argv[1]
    - En caso contrario, imprimimos un error por la salida estándar de error, imprimimos el correcto uso y salimos de la función con un error (1)
  - Control de selección:
    - Comprobamos si el string que ha introducido el usuario se corresponde con alguno de los argumentos permitidos.
      - Si es así, llama a la función correspondiente y almacena su resultado en la variable result
      - Si no, imprimimos el correcto uso y salimos de la función con un error (1)
  - Impresión del resultado
    - Si el resultado es distinto de 0, imprimimos un error y salimos de la función con un error (2)

Principales Funciones

	Main	Nombre	Tipo	Descripción
Argumentos	Argumento 1	argc	int	Número de argumentos que recibe el programa
	Argumento 2	argv	char**	Array que contiene los distintos argumentos
Variables Locales	Variable 1	n	int	Número de líneas especificadas por el usuario
	Variable 2	selection	char*	String que contiene el nombre de la función que el usuario quiere ejecutar
	Variable 3	result	int	Resultado que devuelve la función tras ser ejecutada
Valor Devuelto			int	Resultado de la ejecución
Descripción de la Función	Es el programa principal en el que se ejecuta el test de las funciones.			

	Head	Nombre	Tipo	Descripción
Argumentos	Argumento 1	N	int	Número de líneas
Variables Locales	Variable 1	len	size_t	Longitud máxima de la línea
	Variable 2	buffer	char*	Guarda temporalmente la línea leída
	Variable 3	i	int	Variable auxiliar para iterar
Valor Devuelto			int	Resultado de la ejecución
Descripción de la Función	Emula el mandato `head` de la bash, devuelve por salida estándar las N primeras líneas que recibe por entrada estándar			

	Tail	Nombre	Tipo	Descripción
Argumentos	Argumento 1	N	int	Número de líneas
Variables Locales	Variable 1	len	size_t	Longitud máxima de la línea
	Variable 2	buffer	char*	Guarda temporalmente la línea leída
	Variable 3	i	int	Variable auxiliar para iterar
	Variable 4	data	char**	Guarda los datos resultantes
	Variable 5	current	int	Lleva la posición actual del array
Valor Devuelto			int	Resultado de la ejecución
Descripción de la Función	Emula el mandato `tail` de la bash, devuelve por salida estándar las N últimas líneas que recibe por entrada estándar.			

	Longlines	Nombre	Tipo	Descripción
Argumentos	Argumento 1	N	int	Número de líneas
Variables Locales	Variable 1	len	size_t	Longitud máxima de la línea
	Variable 2	buffer	char*	Guarda temporalmente la línea leída
	Variable 3	i	int	Variable auxiliar para iterar
	Variable 4	data	char**	Guarda los datos resultantes
	Variable 5	count	int	Lleva la posición actual del array
	Variable 6	worstLine	int	Variable auxiliar que guarda la longitud de la línea más corta en Data
	Variable 7	worstLineIndex	int	Variable auxiliar que guarda la posición de la línea más corta en Data
	Variable 8	lengths	int*	guarda las longitudes de las líneas en Data
Valor Devuelto			int	Resultado de la ejecución
Descripción de la Función	Imprime las N líneas más largas recibidas por entrada estándar.			

	CompareStringsByLength	Nombre	Tipo	Descripción
Argumentos	Argumento 1	a	const void *	String a comparar (1)
	Argumento 2	b	const void *	String a comparar (2)
Valor Devuelto			int	Devuelve 0 si la longitud de los strings es igual. Devuelve un número positivo si la longitud de a es mayor a la de b. Devuelve un número negativo en caso contrario.
Descripción de la Función	Compara dos strings por su longitud.			

Casos de Prueba

Se ha utilizado el fichero <data_input.txt> para las distintas pruebas.

Head

Test	Input	Output

1	`./test -head < tests/data_input.txt`	`This is LINE 1__`
		`This is LINE 2`
		`This is LINE 3`
		`This is LINE 4`
		`This is LINE 5`
		`This is LINE 6`
		`This is LINE 7`
		`This is LINE 8`
		`This is LINE 9__`
		`This is LINE 10`

2	`./test -head 3 < tests/data_input.txt`	`This is LINE 1__`
		`This is LINE 2`
		`This is LINE 3`

3	`./test -head 3`	1 1 2 2 3 3

Tail

Test	Input	Output

1	`./test -tail < tests/data_input.txt`	`This is LINE 31`
		`This is LINE 32`
		`This is LINE 33`
		`This is LINE 34`
		`This is LINE 35`
		`This is LINE 36`
		`This is LINE 37`
		`This is LINE 38`
		`This is LINE 39`
		`This is LINE 40`

2	`./test -tail 3 < tests/data_input.txt`	`This is LINE 38`
		`This is LINE 39`
		`This is LINE 40`

3	`./test -tail 3`	`1`
		`2`
		`3`
		`4`
		`5`
		`(Ctrl + D)`
		`3`
		`4`
		`5`

Longlines

Test	Input	Output

1	`./test -longlines < tests/data_input.txt`	`This is LINE 17__`
		`This is LINE 1__`
		`This is LINE 9__`
		`This is LINE 11`
		`This is LINE 12`
		`This is LINE 13`
		`This is LINE 14`
		`This is LINE 15`
		`This is LINE 16`
		`This is LINE 10`

2	`./test -longlines 3 < tests/data_input.txt`	`This is LINE 17__`
		`This is LINE 1__`
		`This is LINE 9__`

3	`./test -longlines 3`	`Lorem Ipsum`
		`---`
		`1234`
		`__asd`
		`(Ctrl + D)`
		`Lorem Ipsum`
		`__asd`
		`1234`

Comentarios Personales

PROBLEMAS ENCONTRADOS

Longlines

En el desarrollo de la función longlines, en una primera aproximación, habíamos decidido implementarla mediante una bicola con puntero al final, sacada de una librería (<sys/queue.h>).

Sin embargo, esta implementación era demasiado compleja para el problema que realmente teníamos, puesto que el tamaño de la estructura es fijo y determinado por el usuario.

Por lo tanto, tras unas violaciones de segmento, decidimos implementar un array circular en el que íbamos sobrescribiendo las líneas más cortas.

Finalmente, después de darle unas vueltas, llegamos a la conclusión de guardar la longitud de las líneas en una estructura auxiliar para evitar calcular la longitud de las líneas constantemente.

Tail

Tras llegar a esta conclusión con longlines, modificamos también la implementación de Tail para que usara un array circular de la misma manera sobrescribiendo las líneas más antiguas.

Para arreglar el orden de impresión, llevamos la variable current, que nos permite ver desde dónde imprimir en este array circular.

CRÍTICAS CONSTRUCTIVAS

Nos ha parecido que la práctica se adapta perfectamente a los conocimientos aprendidos en el tema 3.

Eleva un poco el nivel de los ejercicios, pero con el tiempo que tenemos para realizarla es muy asequible.

PROPUESTA DE MEJORAS

Proponer la implementación de una función que sea un poco diferente al resto, ya que son bastante similares.

EVALUACIÓN DEL TIEMPO DEDICADO

Programación

Hemos dedicado alrededor de 2 tardes para el código y alguna hora extra para optimizarlo con nuevas ideas que se nos han ocurrido a lo largo del tiempo.

Memoria

Hemos dedicado una mañana entera para intentar cumplir de la mejor manera posible todos los requisitos de la memoria.