

Práctica 1

Sistemas

operativos

Jordi Blasco Lozano

Sistemas operativos y distribuidos

Grado en Inteligencia Artificial

Indice:

Indice:	2
1. Ejercicio 1: GESTIÓN BÁSICA CON SCRIPTS	3
2. Ejercicio 2: GESTIÓN BÁSICA DE PROCESOS	5
3. Ejercicio 3: COMUNICACIÓN ENTRE PROCESOS: TUBERÍAS	6
4. Ejercicio 4: COMUNICACIÓN ENTRE PROCESOS: MEMORIA COMPARTIDA	7

1. Ejercicio 1: GESTIÓN BÁSICA CON SCRIPTS

1.1 Filtrar Procesos Activos con Mayor Consumo de Memoria

Comandos usados: ps, sort, head.

Implementación:

Usamos el comando ps -aux para listar todos los procesos.

Luego se ordenan según el uso de memoria con --sort=-%mem.

Se usa head -n N para mostrar los N procesos que más memoria consumen.

Se añade una línea más para la descripción de procesos.

```
1  #!/bin/bash
2
3  # Solicita al usuario el número de procesos que desea ver
4  printf "Indica los procesos que quieras ver: "
5
6  # Lee el valor ingresado por el usuario y lo guarda en la variable N
7  read N
8
9  # Lista todos los procesos del sistema, ordenados por uso de memoria (de mayor a menor)
10 # Luego, selecciona las primeras N+1 líneas para incluir los encabezados
11 ps -aux --sort=-%mem | head -n $((N+1))
```

Resultados:

```
Indica los procesos que quieras ver: 4
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   6052  3724 pts/0    Ss   21:19   0:00 /bin/bash
root        61  0.0  0.0   8648  3260 pts/0    R+   21:36   0:00 ps -aux --sort=-%mem
root        60  0.0  0.0   5788  3136 pts/0    S+   21:36   0:00 /bin/bash ./consumo_proc.sh
root        62  0.0  0.0   4288   560 pts/0    S+   21:36   0:00 head -n 5
root@517ec5e5ebb2:/workdir#
```

(solo tengo 4 procesos activos ya que uso docker)

1.2 Contar Archivos y Directorios en un Directorio

Comandos usados: find, wc, tr

Implementación:

if [\$# -eq 0]: comprueba si no hay argumentos para implementar el código en el directorio actual si no existe el argumento, y en la ruta que le pasemos si sí que existe el argumento.

ruta="\$1": guardamos la ruta en la variable.

find "\$ruta" -type d: usamos el comando find para buscar todos los elementos de tipo directorio en la ruta indicada.

| wc -l : pasamos la salida del comando anterior por este comando que devolverá las líneas que haya ejecutado el comando anterior es decir los directorios.

| tr -d '[:space:]': eliminamos cualquier espacio en blanco de la salida para que salga el número solo.

Lo guardamos en una variable y lo imprimimos, igual que con los archivos.

```
1  #!/bin/bash
2
3  # Si no se proporciona un argumento, se asume que el directorio es el actual
4  if [ $# -eq 0 ]; then
5      ruta="."
6  else
7      ruta="$1"
8  fi
9
10 # Contar el número de archivos en el directorio actual
11 num_archivos=$(find "$ruta" -type f | wc -l | tr -d '[:space:]')
12
13 # Contar el número de directorios en el directorio actual
14 num_directorios=$(find "$ruta" -type d | wc -l | tr -d '[:space:]')
15
16
17 printf "Número de archivos: %d\n" "$num_archivos"
18 printf "Número de directorios: %d\n" "$num_directorios"
19
```

Resultados:

Para probar el programa correctamente al usar docker me copio a tiempo real el directorio que tengo en windows en la carpeta workdir de docker, así que probé a lanzar docker desde una carpeta bastante primitiva a la carpeta donde está el archivo. Y lo que hay en workdir es todo el primer y segundo año, todo lo que ha contado el programa.

```
root@414f9ff96c71:/workdir/2doAnyo# ./sistemasDistribuidos/practica/prac1/contar_archivos.sh /workdir
Número de archivos: 3367
Número de directorios: 429
root@414f9ff96c71:/workdir/2doAnyo# ls
Dockerfile algoritmia build.bat build.sh progAvanzada representacionConocimiento requirements.txt run.bat run.sh sistemasDistribuidos
```

2. Ejercicio 2: GESTIÓN BÁSICA DE PROCESOS

Comandos usados: fork, kill, signal, execvp, wait.

Implementación:

Para este ejercicio, las variables se inicializaron como globales para asegurar que cada proceso pudiera acceder a ellas correctamente, ya que cada proceso generado necesita conocer el estado de las demás variables compartidas.

Se utilizó un manejador de señales para enviar una señal al proceso especificado como primer parámetro en la ejecución. Esto permite coordinar la ejecución y sincronización entre procesos. Además, se emplearon dos **pause()** obligatorios en los procesos **X** y **Y** para evitar que estos procesos terminen antes de tiempo. La razón de estas pausas es asegurar que los procesos **X** y **Y** permanezcan en espera hasta que se les indique que finalicen, lo cual se realiza mediante señales desde el proceso **Z** después de ejecutar sus funciones (**pstree** o **ps**). Esto garantiza que todos los procesos finalicen en el orden correcto.

Resultados:

```
root@6788dd880bca:/workdir# ./ejec1 A 7
senal: A
Soy el proceso ejec: mi pid es 132
Soy el proceso A: mi pid es 133. Mi padre es 132
Soy el proceso B: mi pid es 134. Mi padre es 133. Mi abuelo es 132
Soy el proceso X: mi pid es 135. Mi padre es 134. Mi abuelo es 133. Mi bisabuelo es 132
Soy el proceso Y: mi pid es 136. Mi padre es 134. Mi abuelo es 133. Mi bisabuelo es 132
Soy el proceso Z: mi pid es 137. Mi padre es 134. Mi abuelo es 133. Mi bisabuelo es 132
senal enviada a 133
ls
Enunciados_S0.pdf 'P1 sistemas.docx' Practica1_S0.pdf consumo_proc.sh contar_archiv
Soy X (135) y muero
Soy Y (136) y muero
Soy Z (137) y muero
Soy B (134) y muero
Soy A (133) y muero
Soy ejec (132) y muero
root@6788dd880bca:/workdir# ./ejec1 X 3
senal: X
Soy el proceso ejec: mi pid es 139
Soy el proceso A: mi pid es 140. Mi padre es 139
Soy el proceso B: mi pid es 141. Mi padre es 140. Mi abuelo es 139
Soy el proceso X: mi pid es 142. Mi padre es 141. Mi abuelo es 140. Mi bisabuelo es 139
Soy el proceso Y: mi pid es 143. Mi padre es 141. Mi abuelo es 140. Mi bisabuelo es 139
Soy el proceso Z: mi pid es 144. Mi padre es 141. Mi abuelo es 140. Mi bisabuelo es 139
senal enviada a 142
pstree
bash(1)—ejec1(139)—ejec1(140)—ejec1(141)—ejec1(142)—pstree(145)
                                     —ejec1(143)
                                     —ejec1(144)

Soy X (142) y muero
Soy Y (143) y muero
Soy Z (144) y muero
Soy B (141) y muero
Soy A (140) y muero
Soy ejec (139) y muero
root@6788dd880bca:/workdir#
```

3. Ejercicio 3: COMUNICACIÓN ENTRE PROCESOS: TUBERÍAS

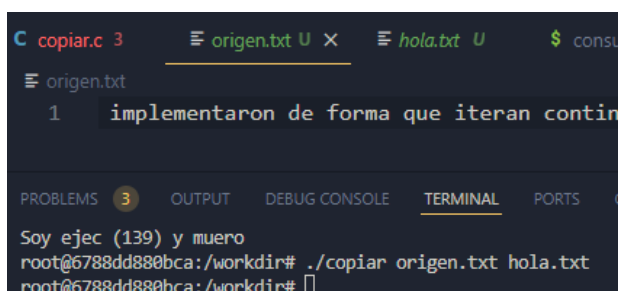
Comandos usados: fork, pipe, read, write, open, close.

Implementación:

Para este ejercicio se declararon las variables necesarias para abrir la tubería, gestionar los archivos, manejar los bytes leídos, los descriptores y el buffer. Se decidió utilizar variables intermedias para almacenar los resultados de las operaciones de apertura, lectura y escritura, en lugar de usarlas directamente en los condicionales. Esto mejora la legibilidad del código y permite una mejor comprensión de cada operación. Los bucles **while** se implementaron de manera que iteran continuamente (**while true**) hasta que se hayan leído y transferido todos los bytes necesarios, utilizando una instrucción **break** para finalizar el bucle de forma controlada una vez completada la transferencia.

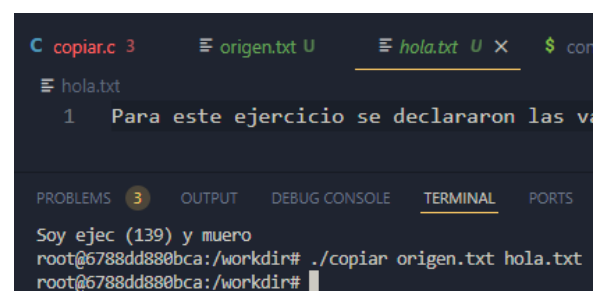
En primer lugar, se abre el archivo origen, se crea una tubería y se genera un proceso hijo a partir del proceso padre. Una vez que el proceso hijo está en ejecución, se cierra el extremo de la tubería, ya que el hijo solo necesita leer de esta. Luego se abre el archivo de destino (si no existe, se crea) y se leen todos los bytes de la tubería mientras el proceso padre los está escribiendo simultáneamente en esta. Conforme se leen de la tubería, se van escribiendo en el archivo destino. El proceso padre funciona de forma inversa: lee del archivo origen y escribe en la tubería. Como indicamos al proceso hijo que termine una vez haya completado su tarea, el hijo no repetirá el trabajo del padre, y ambos bucles se ejecutarán al mismo tiempo.

Resultados:



```
C copiar.c 3  origen.txt U X  hola.txt U  $ cons
1 implementar de forma que iteran contin

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Soy ejec (139) y muero
root@6788dd880bca:/workdir# ./copiar origen.txt hola.txt
root@6788dd880bca:/workdir#
```



```
C copiar.c 3  origen.txt U  hola.txt U X  $ con
1 Para este ejercicio se declararon las v

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Soy ejec (139) y muero
root@6788dd880bca:/workdir# ./copiar origen.txt hola.txt
root@6788dd880bca:/workdir#
```

4. Ejercicio 4: COMUNICACIÓN ENTRE PROCESOS: MEMORIA COMPARTIDA

Comandos usados: fork, shmget, shmat, shmdt, shmctl, ftok.

Implementación:

Tal como hicimos en el ejercicio anterior, declaramos las variables al inicio y utilizamos más variables de las necesarias para facilitar la comprensión del código. De esta manera, podremos usar estas variables dentro de los condicionales para tener claro qué se está haciendo en cada momento del código.

En este ejercicio, vamos a crear un segmento de memoria compartida al cual le asignaremos una variable llamada **nums**, que será un array de números. Crearemos un proceso hijo que generará una semilla para la generación de números aleatorios y, mediante un bucle que llegará hasta el número de valores deseados (en este caso 10), llenará la variable **nums** de la memoria compartida con esos números aleatorios.

Una vez completado, el proceso hijo terminará, mientras el proceso padre, que estará esperando que su hijo finalice, imprimirá los números de la variable **nums** del segmento de memoria compartida. Luego de esto, calculará la media de los números.

Resultados:

```
root@6788dd880bca:/workdir# ./numAleatorios
Soy el hijo (154): los números generados son: 81, 55, 64, 50, 36, 48, 81, 77, 36, 44
Soy el padre (153). Los números generados fueron: 81, 55, 64, 50, 36, 48, 81, 77, 36, 44
La media es de 57.20
root@6788dd880bca:/workdir# ./numAleatorios
Soy el hijo (156): los números generados son: 97, 85, 61, 95, 70, 10, 74, 71, 80, 54
Soy el padre (155). Los números generados fueron: 97, 85, 61, 95, 70, 10, 74, 71, 80, 54
La media es de 69.70
root@6788dd880bca:/workdir#
```