



Sistemas Operativos

Segundo Cuatrimestre de 2020

Proyecto

Comisión:

- Giordano, Gino - 118431
- Seewald Urban, Guillermo - 111699

Índice

| | |
|---|-----------|
| Procesos, threads y comunicación | 4 |
| Conjunto de tareas: | 4 |
| Solución | 4 |
| Algoritmos | 4 |
| Utilización | 8 |
| Mini Shell | 10 |
| Estrategia de solución | 10 |
| Algoritmo | 10 |
| Utilización | 11 |
| mkdir | 11 |
| rmdir | 14 |
| mkfile | 16 |
| lsdir | 18 |
| lsfile | 20 |
| chmod | 21 |
| help | 23 |
| Sincronización | 24 |
| Demasiadas botellas de leche | 24 |
| Demasiadas botellas de leche con semáforos | 24 |
| Solución | 24 |
| Algoritmo | 24 |
| Utilización | 25 |
| Demasiadas botellas de leche con cola de mensajes | 26 |
| Solución | 26 |
| Utilización | 26 |
| Comida rápida | 27 |
| Comida rápida con semáforos | 27 |
| Solución | 27 |
| Algoritmos | 28 |
| Problema en la solución planteada | 29 |
| Comida rápida con cola de mensajes | 29 |
| Solución | 29 |
| Algoritmos | 32 |
| Lectura | 34 |
| Problemas conceptuales | 38 |
| Copy-on-write | 38 |
| FAT | 40 |
| Tablas de información | 42 |

1. Experimentación de Procesos y Threads con los Sistemas Operativos

1. Procesos, threads y comunicación

1. Conjunto de tareas:

Solución

Se tiene un proceso principal el cual lee desde consola la cantidad total de tareas a ejecutar en cada ciclo y se encarga de determinar la cantidad de cada tipo de tarea a ejecutar en base a dicha cantidad total. Una vez que determina la cantidad de cada tipo, se comunica con distintos subprocesos (proceso A, proceso B y proceso C) los cuales están encargados de ejecutar la cantidad específica de un solo tipo de tareas. El proceso A ejecuta x cantidad de tareas de tipo A, el proceso B tareas de tipo B y el proceso C tareas de tipo C. Esta comunicación se realiza a través de pipes, en donde el proceso padre envía a los procesos hijos la cantidad de su tipo de tarea que tienen que ejecutar y además envía los datos adicionales que necesitan cada una de las tareas. Para esta sección se poseen 3 pipes, pipeA, pipeB y pipeC, los cuales son utilizados por el proceso padre para comunicarse con el proceso A, B y C respectivamente.

La forma en la que trabajan estos subprocesos es que, inicialmente, crean una cantidad máxima de hilos en base a la cantidad máxima de tareas de su tipo que se pueden ejecutar en un ciclo. Entre estos hilos y el proceso que los creó se vincula un buffer junto con 3 semáforos. El buffer es utilizado como un medio por el cual el proceso envía el conjunto de datos que una tarea requiere para ejecutar en un ciclo, mientras que los semáforos se utilizan para sincronizar el acceso a dicho buffer y además para sincronizar la finalización de tareas.

Una vez que el proceso detecta que todos los hilos terminaron las tareas, entonces notifica al proceso coordinador este evento a través de un pipe denominado pipeCoordinador. Luego, cuando el proceso coordinador detecta la notificación de finalización de todos los procesos hijos con los cuales se comunicó, da por terminado el ciclo actual y se prepara para repetir el proceso en el siguiente ciclo.

Algoritmos

Proceso coordinador:

---- Inicio algoritmo ----

```
-- iterar
-- cantidad_total <- leer la cantidad ingresada
-- si cantidad_total es igual a 4
    -- indicar al proceso A que ejecute 2 tareas
    -- enviar datos al proceso A para las 2 tareas
    -- indicar al proceso B que ejecute 2 tareas
    -- enviar datos al proceso B para las tareas
    -- esperar a que terminen las tareas de tipo A y tipo B
-- si cantidad_total es igual a 5
    -- indicar al proceso A que ejecute 2 tareas
    -- enviar datos al proceso A para las 2 tareas
    -- indicar al proceso B que ejecute 1 tarea
    -- enviar datos al proceso B para la tarea
    -- indicar al proceso C que ejecute 2 tareas
    -- enviar datos al proceso C para las 2 tareas
    -- esperar a que terminen las tareas de tipo A, B y C
-- si cantidad_total es igual a 6
    -- indicar al proceso A que ejecute 2 tareas
    -- enviar datos al proceso A para las 2 tareas
    -- indicar al proceso B que ejecute 2 tareas
    -- enviar datos al proceso B para las 2 tareas
    -- indicar al proceso C que ejecute 2 tareas
    -- enviar datos al proceso C para las 2 tareas
    -- esperar a que terminen las tareas de tipo A, B y C
```

---- Fin algoritmo ----

Proceso A:

---- Inicio algoritmo ----

```
-- crea un buffer para compartir datos con los hilos
-- crea 3 semáforos para la sincronización, mutex_buffer_A (arranca en 1),
buffer_lleno_A (arranca en 0), tarea_terminada_A (arranca en 0)
-- crear hilos con funcionalidad de tarea de tipo A
-- iterar
    -- espero a que el proceso coordinador indique la cantidad de tareas
    a ejecutar
    -- para i desde 1 hasta la cantidad de tareas a ejecutar
        -- obtengo la información requerida para una tarea A
        -- exclusividad para modificar el buffer | wait(mutex_buffer_A)
```

```
-- escribo en el buffer la información obtenida
-- notifico que escribí en el buffer | signal(buffer_lleno_A)
-- libero exclusividad del buffer | signal(mutex_buffer_A)
-- para i desde 1 hasta la cantidad de tareas a ejecutar
    -- espero a que termine una tarea A | wait(tarea_terminada_A)
-- notificar al proceso coordinador que todas la tareas de tipo A
indicadas finalizaron
    ---- Fin algoritmo ----
```

Proceso B:

```
    ---- Inicio algoritmo ----
-- crea un buffer para compartir datos con los hilos
-- crea 3 semáforos para la sincronización, mutex_buffer_B (arranca en 1),
buffer_lleno_B (arranca en 0), tarea_terminada_B (arranca en 0)
-- crear hilos con funcionalidad de tarea de tipo B
-- iterar
    -- espero a que el proceso coordinador indique la cantidad de tareas
    a ejecutar
    -- para i desde 1 hasta la cantidad de tareas a ejecutar
        -- obtengo la información requerida para una tarea B
        -- exclusividad para modificar el buffer | wait(mutex_buffer_B)
        -- escribo en el buffer la información obtenida
        -- notifico que escribí en el buffer | signal(buffer_lleno_B)
        -- libero exclusividad del buffer | signal(mutex_buffer_B)
    -- para i desde 1 hasta la cantidad de tareas a ejecutar
        -- espero a que termine una tarea B | wait(tarea_terminada_B)
-- notificar al proceso coordinador que todas la tareas de tipo B
indicadas finalizaron
    ---- Fin algoritmo ----
```

Proceso C:

```
    ---- Inicio algoritmo ----
-- crea un buffer para compartir datos con los hilos
-- crea 3 semáforos para la sincronización, mutex_buffer_C (arranca en 1),
buffer_lleno_C (arranca en 0), tarea_terminada_C (arranca en 0)
-- crear hilos con funcionalidad de tarea de tipo C
-- iterar
    -- espero a que el proceso coordinador indique la cantidad de tareas
    a ejecutar
    -- para i desde 1 hasta la cantidad de tareas a ejecutar
        -- obtengo la información requerida para una tarea C
```

```
-- exclusividad para modificar el buffer | wait(mutex_buffer_C)
-- escribo en el buffer la información obtenida
-- notifico que escribí en el buffer | signal(buffer_lleno_C)
-- libero exclusividad del buffer | signal(mutex_buffer_C)
-- para i desde 1 hasta la cantidad de tareas a ejecutar
-- espero a que termine una tarea A | wait(tarea_terminada_C)
-- notificar al proceso coordinador que todas la tareas de tipo C
indicadas finalizaron
---- Fin algoritmo ----
```

Tarea A:

Las tareas A comparten con el proceso A un buffer y 3 semáforos, mutex_buffer_A, buffer_lleno_A y tarea_terminada_A

```
---- Inicio algoritmo ----
-- iterar
-- espero a que haya un dato en el buffer | wait(buffer_lleno_A)
-- exclusividad para acceder al buffer | wait(mutex_buffer_A)
-- obtengo los datos (si la tarea es parcial o total y además el color)
-- libero la exclusividad del buffer | signal(mutex_buffer_A)
-- si el tipo es parcial
-- se realiza tarea de pintado, con 1 unidad de tiempo
-- sino (el tipo es total)
-- se realiza tarea de pintado, con 3 unidades de tiempo
-- notifico que se terminó la tarea | signal(tarea_terminada_A)
---- Fin algoritmo ----
```

Tarea B:

Las tareas B comparten con el proceso B un buffer y 3 semáforos, mutex_buffer_B, buffer_lleno_B y tarea_terminada_B

```
---- Inicio algoritmo ----
-- iterar
-- espero a que haya un dato en el buffer | wait(buffer_lleno_B)
-- exclusividad para acceder al buffer | wait(mutex_buffer_B)
-- obtener datos (si la tarea es verificación o reparación)
-- libero la exclusividad del buffer | signal(mutex_buffer_B)
-- si la tarea es verificación
-- se realiza tarea de verificación, con 1 unidad de tiempo
-- sino
se realiza tarea de reparación, con 2 unidades de tiempo
```

```
-- notifico que se terminó la tarea | signal(tarea_terminada_B)
---- Fin algoritmo ----
```

Tarea C:

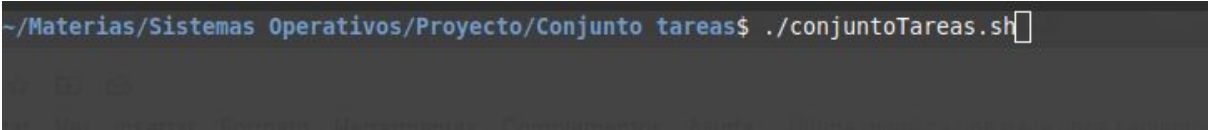
Las tareas A comparten con el proceso A un buffer y 3 semáforos, mutex_buffer_C, buffer_lleno_C y tarea_terminada_C

```
---- Inicio algoritmo ----

-- iterar
    -- espero a que haya un dato en el buffer | wait(buffer_lleno_C)
    -- exclusividad para acceder al buffer | wait(mutex_buffer_C)
    -- obtener datos (si la tarea es reparación o rotación y balanceo, y la
    cantidad de ruedas a reparar)
    -- libero la exclusividad del buffer | signal(mutex_buffer_C)
    -- si la tarea es reparación
        -- se realiza tarea de reparación,, con 1 unidad de tiempo por
        cada rueda
    -- sino
        se realiza tarea de rotación y balanceo, con 3 unidades de
        tiempo
    -- notifico que se terminó la tarea | signal(tarea_terminada_C)
---- Fin algoritmo ----
```

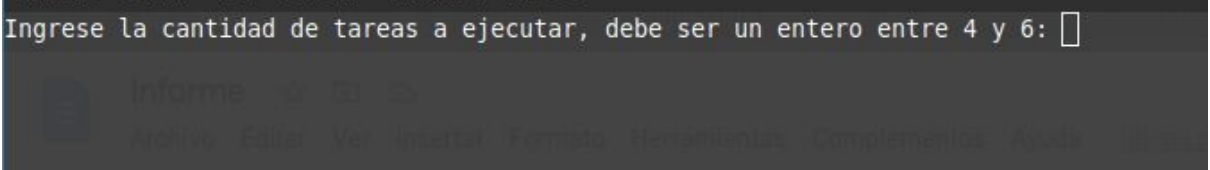
Utilización

Para ejecutar el programa, se debe ejecutar el script denominado *conjuntoTareas.sh* ubicado en la carpeta “/Conjunto tareas”



```
~/Materias/Sistemas Operativos/Proyecto/Conjunto tareas$ ./conjuntoTareas.sh
```

Una vez ejecutado, comenzará la ejecución del programa principal inicializandose tal como se muestra en la siguiente imagen, en donde se podrá seleccionar la cantidad de tareas que se desean ejecutar en el ciclo actual.



```
Ingrese la cantidad de tareas a ejecutar, debe ser un entero entre 4 y 6: 
```


Cuando la opción deseada es ingresada, se comenzará el trabajo indicado según la cantidad de tareas brindada, tal como se muestra a continuación con un ejemplo de 4 tareas:

```
Ingrese la cantidad de tareas a ejecutar, debe ser un entero entre 4 y 6: 4
-----
Ejecutando 4 tareas
  TAREA A: Pintando de color violeta, el trabajo es parcial, requiere una unidad de tiempo
  TAREA A: Pintando de color verde, el trabajo es parcial, requiere una unidad de tiempo
  TAREA B: Realizando verificación de frenos, requiere una unidad de tiempo
  TAREA B: Realizando verificación de frenos, requiere una unidad de tiempo
  Terminó una TAREA A
  Terminó una TAREA A
  Terminó una TAREA B
  Terminó una TAREA B
Terminaron las 4 tareas
-----
¿Desea seguir ejecutando? Ingrese <0> si desea terminar u otro caracter si desea continuar: [ ]
```

Al finalizar el ciclo, se mostrará un mensaje preguntando si desea finalizar el programa o continuar con un nuevo ciclo con una nueva cantidad de tareas a ejecutar. En caso de ingresar 0, el programa finalizará, caso contrario (otro caracter), el programa volverá a preguntar por la cantidad de tareas que se desean ejecutar repitiendo así el proceso anterior.

2. Mini Shell

Estrategia de solución

Se tiene un proceso principal encargado del procesamiento de los comandos ingresados desde la consola y de separar los diferentes argumentos. Una vez que han sido separados, se crea un nuevo proceso y se le carga la imagen que contiene la secuencia de instrucciones del comando solicitado. En caso de que el comando no exista, simplemente se notifica dicho suceso. En caso contrario, el proceso hijo ejecutará las instrucciones de dicho comando y el proceso principal esperará la finalización del proceso hijo antes de proceder a la lectura del siguiente comando. Para realizar la funcionalidades de los diferentes comandos se hizo uso de diferentes librerías provista por el sistema operativo.

Algoritmo

```
----- Inicio algoritmo -----  
mientras no finalizó  
    leído <- leer texto ingresado  
    argumentos <- separarArgumentos(leído)  
    si el primer argumento no es finalizar  
        crear un nuevo proceso  
        si soy el nuevo proceso  
            cargar una nueva imagen en el proceso en base  
            a el nombre del comando (primer argumento)  
        sino  
            espero a que finalice el nuevo proceso  
    sino  
        finalizo <- verdadero  
----- Fin algoritmo -----
```

separarArgumentos:

Se encarga de separar una cadena de texto recibida como parámetro en los diferentes argumentos de un comando. El primer argumento va a determinar el nombre del comando, mientras que los argumentos restantes van a ser cada uno de los argumentos requeridos y opcionales que serán enviados adicionalmente al comando a ejecutar. La separación se realiza en base a los espacios del texto, que indican las diferentes palabras, excepto en el caso donde las palabras separadas por espacios se encuentren encerrados entre comillas dobles, lo cual indicaría que todo el conjunto de palabras es un único argumento.

Utilización

Para utilizar la mini shell, se debe ejecutar el script denominado “minishell.sh”, el cual simplifica la utilización compilando primero todos los archivos fuentes necesarios y luego ejecutando la mini shell.

```
~/Escritorio/Minishell$ ./minishell.sh
```

Una vez ejecutado el script, se abrirá la terminal para el ingreso de comandos.

```
-----  
Bienvenido a la mini shell  
-----  
/home/guille/Escritorio/Minishell $
```

Entre los diferentes comandos disponibles se encuentran:

- **mkdir**: Crea un directorio
- **rmdir**: Elimina un directorio
- **mkfile**: Crea un archivo
- **lsdir**: Lista el contenido de un directorio
- **lsfile**: Muestra el contenido de un archivo
- **chmod**: Modifica los permisos de un archivo
- **help**: Muestra una ayuda con los comandos disponibles
- **exit**: Finaliza y cierra la mini shell

mkdir

Creación de un directorio.

- **Sinopsis**:
 - `mkdir directory [mode]`
 - `mkdir -help`
- **Descripción**

Se presentan dos opciones:

Por un lado, se puede consultar mediante `-help` información acerca del comando.

Por otro lado, ejecuta la función principal del comando, en donde se intenta la creación de un directorio con nombre *directory* y permisos *mode*.

- *directory*:

El formato de *directory* se trata de una cadena de texto, en donde se puede establecer solo el nombre que tendrá el directorio (se creará en la carpeta de ejecución) ó la ruta donde desea crearse junto con el nombre. Si el nombre de algún directorio dentro de la ruta especificada o el propio nombre del directorio a crear contienen más de una palabra, entonces este conjunto de palabras debe encerrarse entre comillas dobles para que sea correctamente interpretado como un único nombre.

- *mode*:

El formato de *mode* se trata de un número entero el cual determina los permisos que tendrá el nuevo directorio. Este argumento es opcional, y en caso de no especificarse se establecerá como predeterminado permisos 0777. Los posibles formatos de este argumento se encuentran determinados por una combinación de 4 bits, $x_1x_2x_3x_4$.

Valores para el segundo bit:

- 0700 El dueño tiene permisos de lectura, escritura y ejecución.
- 0400 El dueño tiene permisos de lectura.
- 0200 El dueño tiene permisos de escritura.
- 0100 El dueño tiene permisos de ejecución

Valores para el tercer bit:

- 0070 El grupo tiene permisos de lectura, escritura y ejecución.
- 0040 El grupo tiene permiso de lectura.
- 0020 El grupo tiene permisos de escritura.
- 0010 El grupo tiene permiso de ejecución.

Valores para el cuarto bit:

- 0007 Otros tienen permisos de lectura, escritura y ejecución (quienes no están en el grupo).
- 0004 Otros tienen permisos de lectura.
- 0002 Otros tienen permisos de escritura.
- 0001 Otros tienen permisos de ejecución.

- *Errores*:

Además de los errores identificados por la librería utilizada (ver `mkdir(2)`), se realiza un control sobre otros errores adicionales dentro del programa de `mkdir`.

- MISSING_ARGUMENTS: En caso de que la cantidad mínima de argumentos necesarios no haya sido alcanzada.
 - EXCEEDED_ARGUMENTS: En caso de que la cantidad máxima de argumentos válidos haya sido sobrepasada.
 - INVALID_ARGUMENTS: En caso de que el *mode* especificado no sea un número entero positivo.
- Ejemplos de secuencias de ejecución:

- Ejecución con argumentos faltantes:

```
/home/guille/Escritorio/Minishell $ mkdir
Faltan argumentos, utilice -help para más información

/home/guille/Escritorio/Minishell $
```

- Ejecución con argumentos de más:

```
/home/guille/Escritorio/Minishell $ mkdir arg1 arg2 arg3
Demasiados argumentos, utilice -help para más información

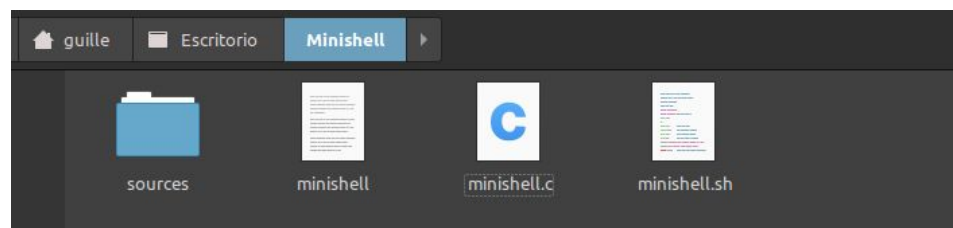
/home/guille/Escritorio/Minishell $
```

- Ejecución con un *mode* inválido (no numérico):

```
/home/guille/Escritorio/Minishell $ mkdir dir1 04modeNoNumerico44
Valor para el parámetro mode no válido, debe ser un entero positivo

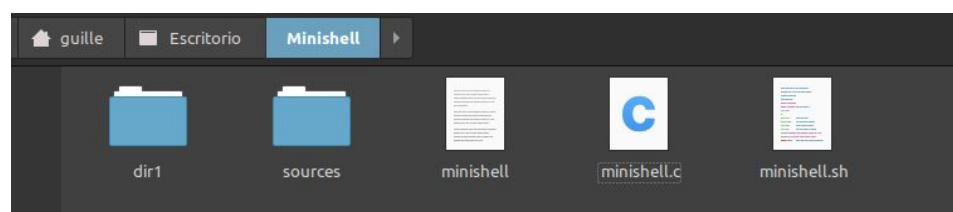
/home/guille/Escritorio/Minishell $
```

- Creación exitosa de un nuevo directorio



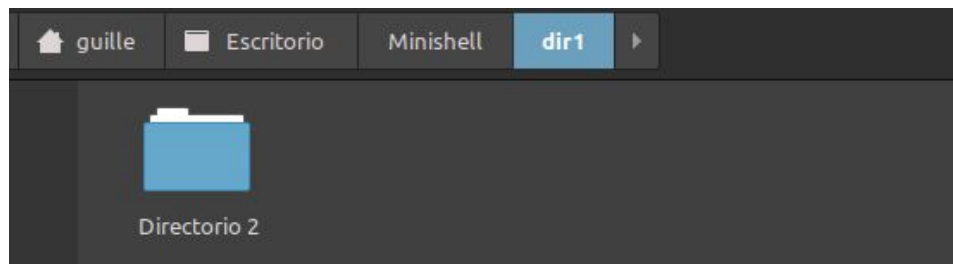
```
/home/guille/Escritorio/Minishell $ mkdir dir1 0777
Creando directorio <dir1> con permisos <777>
Directorio <dir1> creado correctamente

/home/guille/Escritorio/Minishell $
```



```
/home/guille/Escritorio/Minishell $ mkdir dir1/"Directorio 2" 0777
Creando directorio <dir1/Directorio 2> con permisos <777>
Directorio <dir1/Directorio 2> creado correctamente

/home/guille/Escritorio/Minishell $
```



- Creación de un directorio ya existente

```
/home/guille/Escritorio/Minishell $ mkdir dir1 0777
Creando directorio <dir1> con permisos <777>
Error al crear directorio: File exists

/home/guille/Escritorio/Minishell $
```

rmdir

Eliminación de un directorio.

- *Sinopsis*
 - `rmdir directory`
 - `rmdir -help`
- *Descripción*

Se presentan dos opciones:

Por un lado, se puede consultar información acerca del comando mediante el argumento `-help`.

La segunda opción corresponde a la función principal del comando. El argumento *directory* permite especificar el directorio que se desea borrar, ya sea indicando únicamente su nombre (se intentará borrar en la carpeta actual de ejecución) o la ruta completa en la que se encuentra. En caso de que el nombre de uno de los directorios dentro de la ruta o el propio nombre del directorio a eliminar contenga un nombre con más de una palabra se debe establecer el conjunto de palabras entre comillas dobles.
- Errores

Además de los errores identificados por la librería utilizada (ver `rmdir(2)`), se realiza un control sobre otros errores adicionales dentro del programa de `rmdir`.

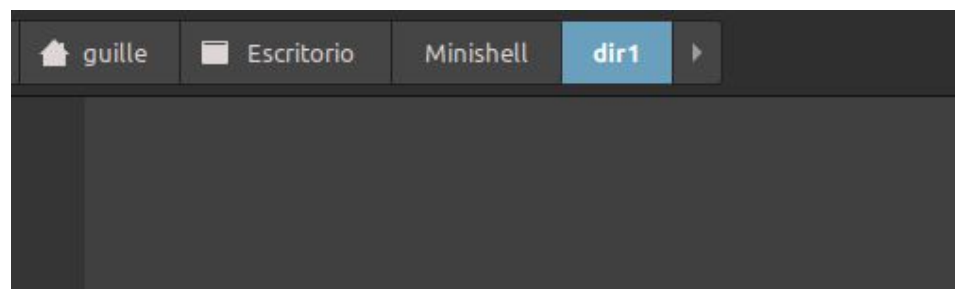
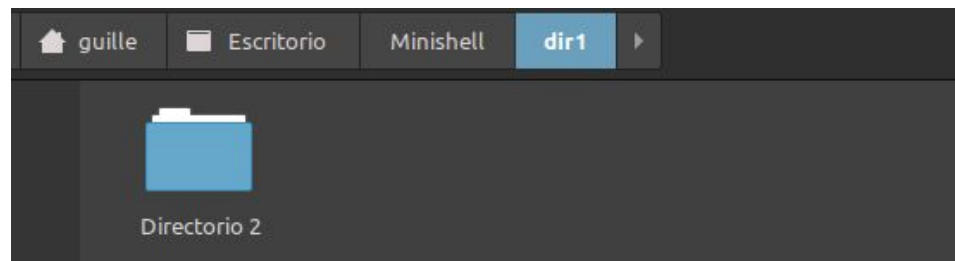
- `MISSING_ARGUMENTS`: En caso de que la cantidad mínima de argumentos necesarios no haya sido alcanzada.
- `EXCEEDED_ARGUMENTS`: En caso de que la cantidad máxima de argumentos válidos haya sido sobrepasada.

- Ejemplos de ejecución

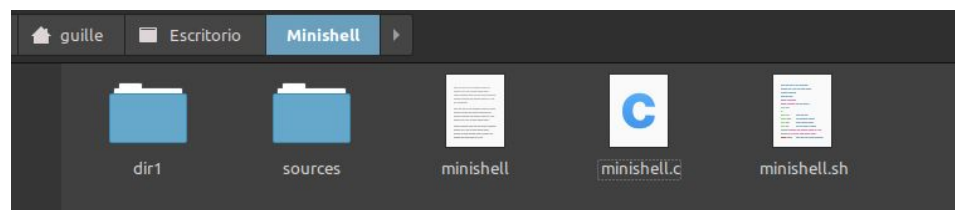
- Ejecución con cantidad máxima de argumentos excedida

```
/home/guille/Escritorio/Minishell $ rmdir arg1 arg2
Cantidad de argumentos máxima excedida, utilice -help para más información
/home/guille/Escritorio/Minishell $
```

- Eliminación exitosa de un directorio



- Eliminación de un directorio no existente



```
/home/guille/Escritorio/Minishell $ rmdir dir2
Error al eliminar el directorio: No such file or directory
/home/guille/Escritorio/Minishell $
```

mkfile

Creación de un archivo

- *Sinopsis*

- `mkfile filename [mode]`
- `mkfilr -help`

- *Descripción*

Se presentan dos opciones:

Por un lado, la opción *-help* muestra una ayuda con información acerca del comando.

La segunda alternativa, la cual modela el comportamiento principal de comando, posee dos argumentos:

- *filename*:

Representa el nombre del archivo a crear. Puede agregar adicionalmente la ruta completa donde se desea que se cree el archivo en caso de que no se desee crearlo en la carpeta actual de ejecución. Si el nombre del archivo o de uno de los directorios de la ruta especificada poseen espacios (más de una palabra), entonces el conjunto de palabras debe ser especificado entre comillas dobles para que sean interpretadas como una única unidad.

- *mode*:

Este argumento es opcional. Permite definir los permisos que tendrá el archivo (ver descripción de `mkdir` para más información acerca de permisos). En caso de que este argumento no sea especificado, se establecerá por defecto permisos `0777`.

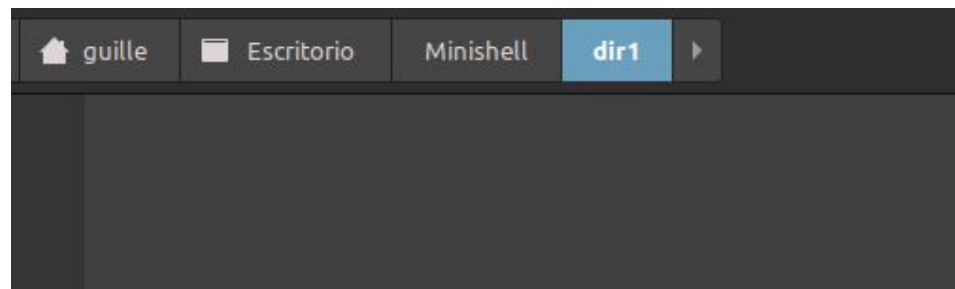
- *Errores*

Además de los errores identificados por la librería utilizada (ver `create(2)`), se realiza un control sobre otros errores adicionales dentro del programa de `mkdir`.

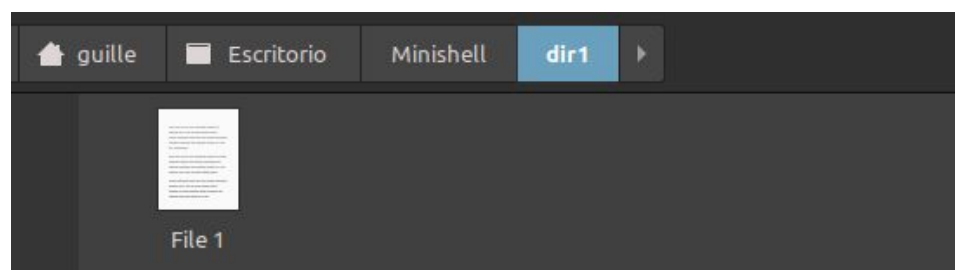
- `MISSING_ARGUMENTS`: En caso de que la cantidad mínima de argumentos necesarios no haya sido alcanzada.
- `EXCEEDED_ARGUMENTS`: En caso de que la cantidad máxima de argumentos válidos haya sido sobrepasada.
- `INVALID_ARGUMENTS`: En caso de que el *mode* especificado no sea un número entero positivo.

- *Ejemplos de ejecución*

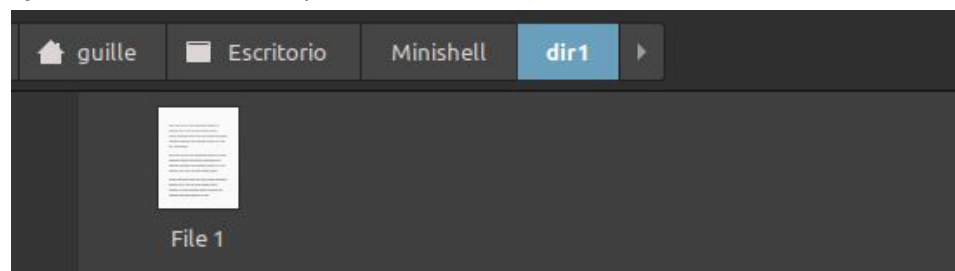
- Ejecución exitosa sin especificar el mode:



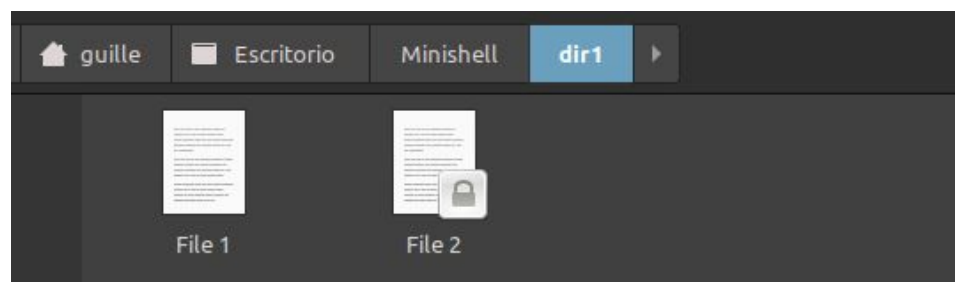
```
/home/guille/Escritorio/Minishell $ mkfile dir1/"File 1"  
Creando archivo <dir1/File 1> con permisos <777>  
Archivo <dir1/File 1> creado correctamente  
  
/home/guille/Escritorio/Minishell $
```



- Ejecución exitosa especificando un *mode* válido:



```
/home/guille/Escritorio/Minishell $ mkfile dir1/"File 2" 0444  
Creando archivo <dir1/File 2> con permisos <444>  
Archivo <dir1/File 2> creado correctamente  
  
/home/guille/Escritorio/Minishell $
```



- Ejecución no exitosa al especificar un *mode* no numérico:

```
/home/guille/Escritorio/Minishell $ mkfile dir1/"File 3" 04modeNoNumerico3
Valor para el parámetro mode no válido, debe ser un entero positivo

/home/guille/Escritorio/Minishell $
```

Isdir

Listado del contenido de un directorio

- *Sinopsis*

- Isdir [directory]
- Isdir -help

- *Descripción*

Se presentan dos opciones de ejecución.

Por un lado, la opción *-help* muestra información acerca del comando.

La segunda opción permite realizar la ejecución principal del comando, en donde se lista un directorio. El único argumento adicional es el de *directory*, el cual especifica la ruta del directorio que se busca listar. Este argumento es opcional y, en caso de no especificarse, Isdir procederá a listar el contenido de la carpeta de ejecución (donde se encuentra actualmente posicionada la mini shell). En caso de que el nombre de uno de los directorios en la ruta contenga espacios (formado por más de una palabra), su nombre debe ser especificado entre comillas dobles.

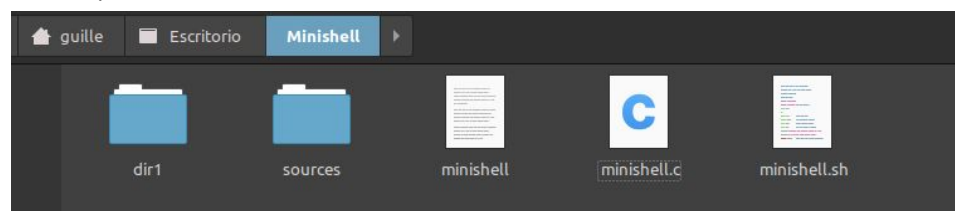
- *Errores*

Además de los errores detectados por las funciones de librería utilizadas (ver `opendir(3)` y `readdir(3)`), se controlan 1 error adicional:

- EXCEEDED_ARGUMENTS: En caso de que la cantidad de argumentos indicada supere la cantidad máxima posible.

- *Ejemplos de ejecución*

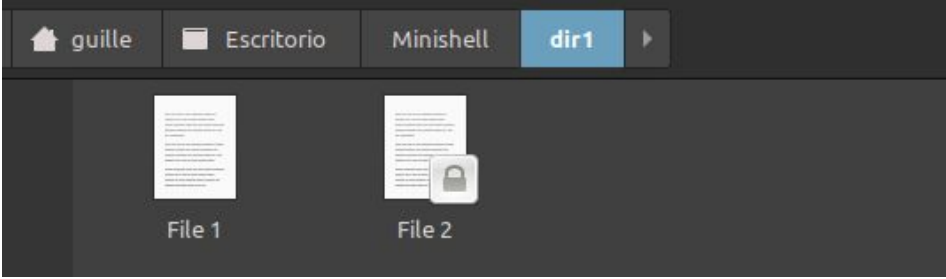
- Ejecución exitosa sin argumentos adicionales (lista directorio actual):



```
/home/guille/Escritorio/Minishell $ lsdir
Contenido del directorio:
1.  dir1
2.  sources
3.  minishell.sh
4.  minishell.c
5.  minishell

/home/guille/Escritorio/Minishell $
```

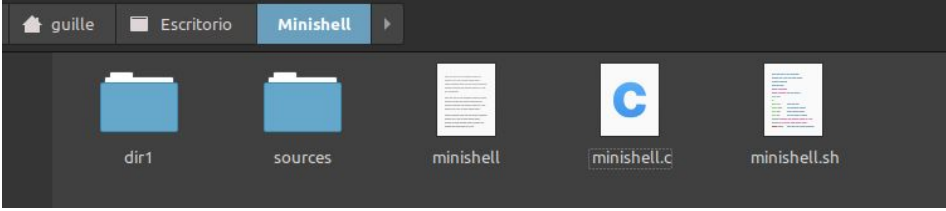
- Ejecución exitosa especificando un directorio:



```
/home/guille/Escritorio/Minishell $ lsdir dir1
Contenido del directorio:
1.  File 2
2.  File 1

/home/guille/Escritorio/Minishell $
```

- Ejecución no exitosa al intentar listar un directorio no existente:



```
/home/guille/Escritorio/Minishell $ lsdir dir2
No se pudo listar el directorio: No such file or directory

/home/guille/Escritorio/Minishell $
```

lsfile

Visualización del contenido de un archivo

- *Sinopsis*
 - lsfile filename
 - lsfile -help

- *Descripción*

Se presentan dos opciones:

La opción -help permite obtener información acerca del comando lsfile.

La opción restante permite realizar el trabajo principal del comando, en donde se muestra el contenido de un archivo el cual es identificado a través del argumento *filename*, donde este puede ser o bien solo el nombre del archivo (se busca en la carpeta de ejecución) o bien la ruta completa donde se encuentra el archivo. En caso de que el nombre del archivo o el de uno de los directorios en la ruta brindada se encuentre formada por más de una palabra, estas deben ser especificadas entre comillas dobles.

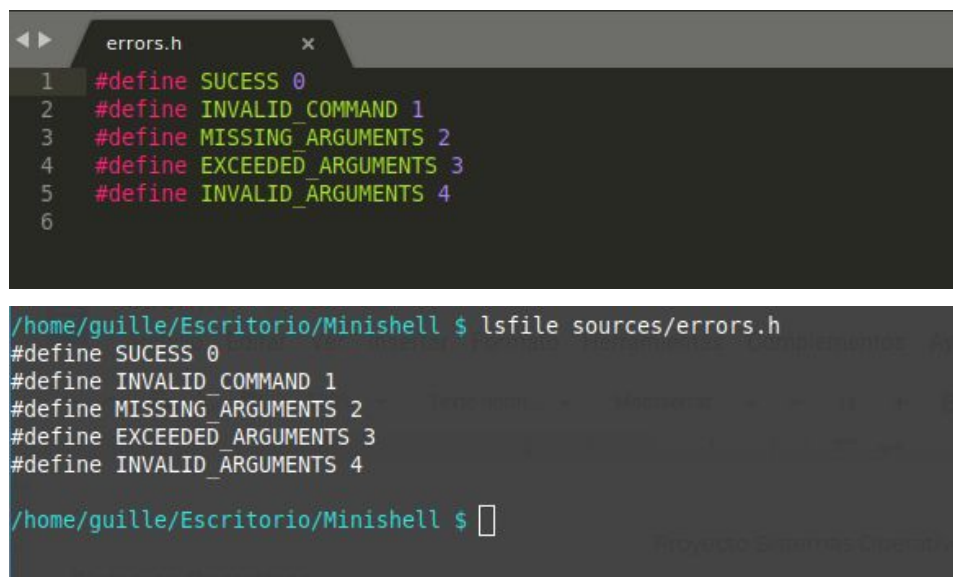
- *Errores*

Además de los errores identificados por las librerías utilizadas (ver fopen(3), geline(3) y fclose(3)), se realiza un control sobre otros errores adicionales dentro del programa de mkdir.

- MISSING_ARGUMENTS: En caso de que la cantidad mínima de argumentos necesarios no haya sido alcanzada.
- EXCEEDED_ARGUMENTS: En caso de que la cantidad máxima de argumentos válidos haya sido sobrepasada.

- *Ejemplos de ejecución*

- Ejecución exitosa



The image shows a code editor window titled 'errors.h' with the following content:

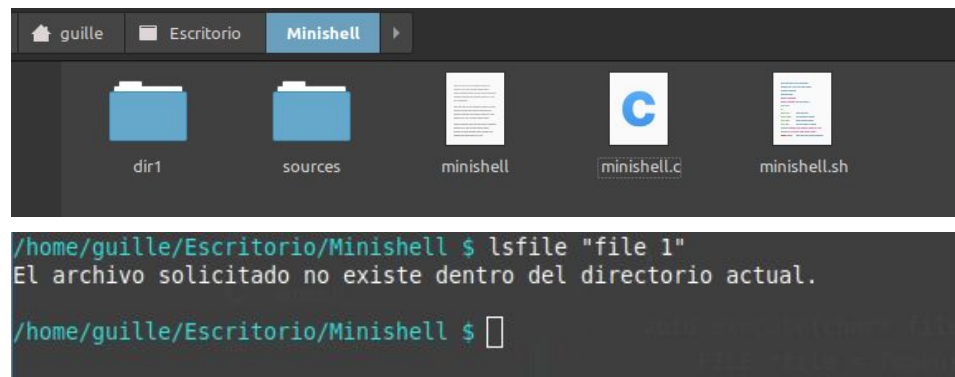
```
1 #define SUCESS 0
2 #define INVALID_COMMAND 1
3 #define MISSING_ARGUMENTS 2
4 #define EXCEEDED_ARGUMENTS 3
5 #define INVALID_ARGUMENTS 4
6
```

Below the code editor is a terminal window showing the command being executed and its output:

```
/home/guille/Escritorio/Minishell $ lsfile sources/errors.h
#define SUCESS 0
#define INVALID_COMMAND 1
#define MISSING_ARGUMENTS 2
#define EXCEEDED_ARGUMENTS 3
#define INVALID_ARGUMENTS 4

/home/guille/Escritorio/Minishell $
```

- Ejecución no exitosa con un archivo no existente



chmod

Modificación de los permisos de un archivo

- Sinópsis
 - `chmod filename mode`
 - `chmod -help`
- Descripción

Se presentan dos opciones:

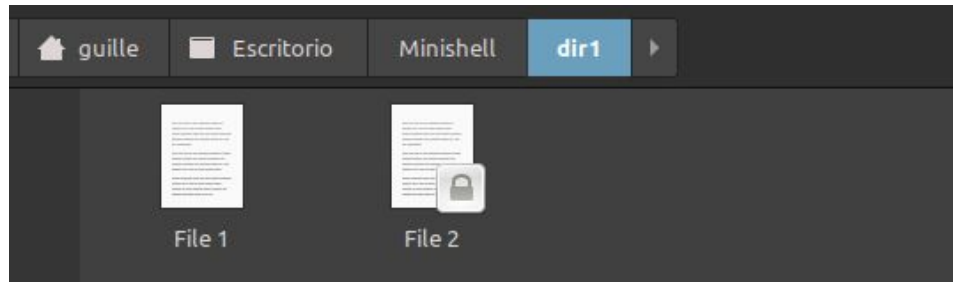
Por un lado, se encuentra la opción de `-help`, la cual permite obtener información acerca del comando.

Por otro lado, se tiene la opción que brinda la funcionalidad principal del comando, en donde se realiza el cambio de permisos de un archivo. El argumento *filename* especifica la ruta del archivo que desea modificarse, mientras que el argumento *mode* establece los nuevos permisos que van a ser aplicados al archivo (para especificaciones de los diferentes permisos, ver sección de `mkdir`). En caso de que un archivo dentro de la ruta especificada del archivo tenga como nombre un conjunto de más de una palabra, estas deben ser especificadas entre comillas dobles para que su nombre sea correctamente identificado.
- Errores

Además de los errores identificados por las librerías utilizadas (ver `fopen(3)`, `getline(3)` y `fclose(3)`), se realiza un control sobre otros errores adicionales dentro del programa de `mkdir`.

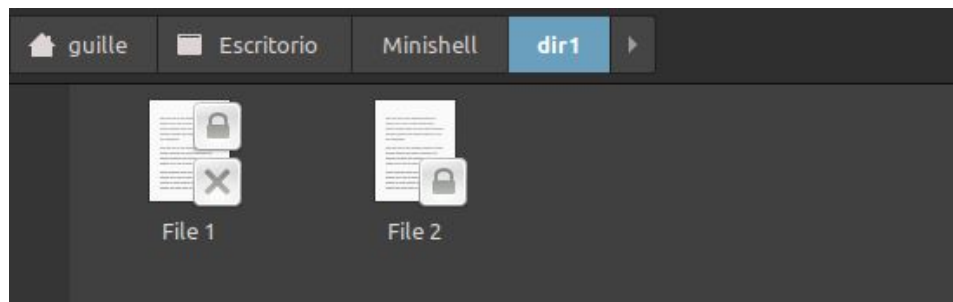
 - `MISSING_ARGUMENTS`: En caso de que la cantidad mínima de argumentos necesarios no haya sido alcanzada.
 - `EXCEEDED_ARGUMENTS`: En caso de que la cantidad máxima de argumentos válidos haya sido sobrepasada.

- INVALID_ARGUMENTS: En caso de que el *mode* especificado no sea un número entero positivo.
- Ejemplos de ejecución
 - Ejecución exitosa:

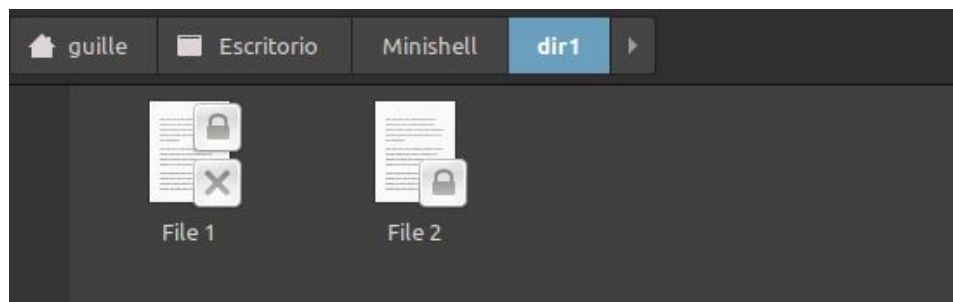


```
/home/guille/Escritorio/Minishell $ chmod dir1/"File 1" 0111
Modificando el archivo <dir1/File 1>, estableciendo permisos <111>
Permisos modificados correctamente para el archivo dir1/File 1

/home/guille/Escritorio/Minishell $
```



- Ejecución no exitosa debido a que el archivo especificado no existe:



```
/home/guille/Escritorio/Minishell $ chmod dir1/"File 3" 0777
Modificando el archivo <dir1/File 3>, estableciendo permisos <777>
Error al modificar los permisos: No such file or directory

/home/guille/Escritorio/Minishell $
```

- Ejecución no exitosa debido a la falta de argumentos:

```
/home/guille/Escritorio/Minishell $ chmod dir1/"File 1"  
Argumentos faltantes, utilice -help para más información  
  
/home/guille/Escritorio/Minishell $
```

- Ejecución no exitosa debido a que el *mode* especificado no es un valor válido:

```
/home/guille/Escritorio/Minishell $ chmod dir1/"File 1" 12modeNoNumerico  
Valor para el parámetro mode no válido, debe ser un entero positivo  
  
/home/guille/Escritorio/Minishell $
```

help

Listado de los comandos disponibles

- *Sinópsis*
 - help
- *Ejemplo de ejecución*

```
/home/guille/Escritorio/Minishell $ help  
Comandos disponibles:  
1. mkdir    Crea un directorio  
2. rmdir    Eliminar un directorio  
3. mkfile   Crea un archivo  
4. lsdir    Lista el contenido de un directorio  
5. lsfile   Muestra el contenido de un archivo  
6. chmod    Modifica los permisos de un archivo  
7. help     Muestra esta ayuda y finaliza  
7. exit     Finaliza la ejecución de la minishell  
  
Utilice <nombre_de_comando> -help para obtener más información de un determinado comando  
  
/home/guille/Escritorio/Minishell $
```

2. Sincronización

1. Demasiadas botellas de leche

a) Demasiadas botellas de leche con semáforos

Solución

Para la solución de este problema, se planteó el uso de 4 semáforos.

Uno de ellos, **heladera** cumple la función de que un compañero no pueda abrir la heladera si otro ya está haciendo uso de la misma, por lo que deberá esperar a que esta sea liberada. Cuando un compañero abre la heladera se deberá fijar si quedan botellas. Para esto se utilizó el semáforo **botellas** que contabiliza la cantidad, el cual comienza con una cantidad mayor a cero (en este caso 10). En caso de que el compañero logre tomar una botella, luego se fija si quedan más botellas a parte de la que ya tomó. Si no quedan más botellas adicionales, significa que debe ir a comprarse más. Para esto, se hace uso de los dos semáforos restantes. Uno de ellos es el denominado **avisoComprar**, el cual simboliza la notificación de que hay que comprar más botellas. El otro, denominado **mutex_avisoComprar**, simboliza la exclusividad para la notificación de que hay que comprar más botellas. Una vez que se haya usado la heladera esta se libera y se continúa con la ejecución

Por otro lado el compañero deberá verificar si se notificó que se debe comprar más botellas. En este caso, se utilizan nuevamente los semáforos **mutex_avisoComprar** y **avisoComprar**. Primero, deberá tener exclusividad para poder verificar que se haya notificado **avisoComprar**. Si se notificó, entonces el compañero que detectó esto deberá ir a comprar botellas de leches y una vez que vuelva con las botellas de leche deberá hacer uso de la heladera para reponer las mismas. Para ello, se volverá a hacer uso del semáforo **heladera** para coordinar la reposición de las botellas y a su vez se incrementará el valor del semáforo de **botellas** a su máximo (en este caso 10).

Algoritmo

Compañero:

---- Inicio algoritmo ----

-- iterar

-- exclusividad en la heladera | wait(heladera)

-- si hay botellas en la heladera | waittry(botellas)

 tomo una botella

-- si hay más botellas aparte de la que saqué | waittry(botellas)

 -- devuelvo la extra que saqué | signal(botellas)

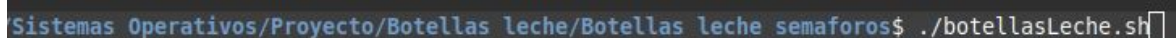

```
-- sino (no quedan más botellas, fui quien sacó la última)
-- exclusividad de panel de avisos |
wait(mutex_avisoComprar)
-- deajo un aviso de que hay que comprar |
signal(avisoComprar)
-- libero la exclusividad del panel de avisos |
signal(mutex_avisoComprar)
-- libero la exclusividad de la heladera | signal(heladera)

-- exclusividad panel de notas | wait(mutex_avisoComprar)
-- si hay un aviso | waittry(avisoComprar)
-- voy a comprar
-- exclusividad de la reponer la heladera | wait(heladera)
-- para i desde 1 hasta cantidad máxima de botellas
-- reponer botellas | signal(botellas)
-- libero exclusividad de la heladera | signal(heladera)
-- libero exclusividad panel de avisos | signal(mutex_avisoComprar)

---- Fin algoritmo ----
```

Utilización

Para utilizar el programa, debe ejecutarse el script denominado *botellasLeche.sh* ubicado en la ruta “/Botellas leche/Botellas leche semaforos”.



```
Sistemas Operativos/Proyecto/Botellas leche/Botellas leche semaforos$ ./botellasLeche.sh
```

Una vez ejecutado, comenzará inmediatamente la ejecución del programa

b) Demasiadas botellas de leche con cola de mensajes

Solución

Para realizar la solución de este problema, se reutilizó el algoritmo planteado en la solución anterior con semáforos y se traspasó el comportamiento hacia una cola de mensajes convirtiendo los distintos semáforos en distintos tipos de mensajes para trabajar con una cola de mensajes. Se reemplazan los semáforos **heladera**, **botellas**, **mutex_avisoComprar** y **avisoComprar** con los tipos **TIPO HELADERA**, **TIPO BOTELLAS**, **TIPO MUTEX COMPRAR** y **TIPO COMPRAR** respectivamente.

Utilización

Para hacer uso del programa, debe ejecutarse el script denominado *botellasLeche.sh* ubicado en la ruta “/Botellas leche/Botellas de leche cola”.

```
Sistemas Operativos/Proyecto/Botellas leche/Botellas de leche cola$ ./botellasLeche.sh
```

Una vez ejecutado el script, se iniciará el programa principal tal como se muestra a continuación, permitiendo ingresar la cantidad de compañeros involucrados, e inmediatamente comenzarán a ejecutarse los diferentes compañeros.

```
-----  
Este programa genera una cantidad N de compañeros ingresado por el usuario para el problema  
"Demasiadas botellas de leche" utilizando cola de mensajes para la sincronización.  
-----  
Ingrese la cantidad de compañeros: 
```

2. Comida rápida

I. Comida rápida con semáforos

Solución

Para la solución de este problema, se planteó el uso de 6 semáforos.

Dos de ellos, **mesaLimpia** y **mesaSucia**, cumplen el propósito de coordinar los clientes y el limpiador. Un cliente no puede ocupar una mesa a menos que ésta se encuentre limpia, por lo que la primera tarea que realiza es esperar hasta que haya una. Al momento de terminar de comer e irse, el cliente estará dejando de ocupar la mesa, pero ésta se encontrará sucia. Para que un nuevo cliente pueda ocupar esta mesa, primero debe limpiarse, lo cual es tarea del limpiador. El limpiador estará esperando hasta que haya una mesa sucia para luego limpiarla y notificar que un nuevo cliente puede ahora ocupar la mesa. Inicialmente, **mesaLimpia** comienza con una cantidad mayor a 0 (en este caso implementado 30), ya que al comenzar ninguna mesa a sido utilizada, y por lo tanto todas se encuentran limpias para que los clientes puedan ocuparlas.

Otros dos semáforos, **comidaPedida** y **comidaEntregada**, son utilizados para la coordinación entre los clientes y el camarero. Una vez que el cliente logró ocupar una mesa, realiza un pedido de comida y luego se queda esperando hasta que el pedido le es entregado. Por otra parte, el camarero realiza la contraparte, ya que estará encargado de esperar a que un cliente realice un pedido, y únicamente cuando esto suceda, entonces obtendrá la comida solicitada y se la brindará al cliente.

Los dos semáforos restantes, **colaDeComidaVacía** y **colaDeComidaLlena**, son utilizados para la sincronización entre el camarero y los cocineros. Para esto hay que tener en cuenta que la cola de comida tiene una capacidad máxima (solo se puede tener preparadas hasta 10 comidas).

Inicialmente, la cola de comida se encuentra vacía, por lo cual los cocineros pueden ir preparando comidas siempre que haya capacidad. Para esto, primero esperan por **colaDeComidaVacía** (esperan a que haya lugar) y una vez que detectan el lugar en la cola de comidas, preparan una comida y la dejan en la cola de comidas, notificando esto a través de **colaDeComidaLlena**.

Por otro lado, los camareros, al momento en que detectan que un cliente realizó un pedido, necesita buscar el pedido de la cola de comidas para poder entregarlo al cliente. Para realizar esto, espera por **colaDeComidaLlena** (el cual indica si hay o no comidas preparadas) y una vez detectado notifica a los cocineros que sacó una comida de la cola de

comida (mediante **colaDeComidaVacía**) para que estos puedan preparar luego una nueva comida y así llenar el hueco que se liberó.

Algoritmos

CLIENTE:

```
---- Inicio algoritmo ----
-- iterar
-- wait(mesaLimpia) | Se espera a que haya una mesa limpia
-- signal(comidaPedida) | Se realiza un pedido de comida
-- wait(comidaEntregada) | Se espera a recibir la comida
-- Comer
-- signal(mesaSucia) | Se deja la mesa sucia
---- Fin algoritmo ----
```

COCINERO:

```
---- Inicio algoritmo ----
-- iterar
-- wait(colaDeComidaVacía) | Se espera a que haya espacio en
la cola de comida
-- preparar comida
-- signal(colaDeComidaLlena) | Se notifica que se preparó una
comida
---- Fin algoritmo ----
```

CAMARERO:

```
---- Inicio algoritmo ----
-- iterar
-- wait(comidaPedida) | Se espera a que haya un pedido
-- wait(colaDeComidaLlena) | Se espera a que haya una
comida preparada
-- signal(colaDeComidaVacía) | Se notifica que se sacó una
comida de la cola de comida
-- signal(comidaEntregada) | Se entrega un pedido a un
cliente
---- Fin algoritmo ----
```

LIMPIADOR:

```
---- Inicio algoritmo ----
-- iterar
-- wait(mesaSucia) | Se espera a que haya una mesa sucia
-- limpiar mesa
-- signal(mesaSucia) | Se notifica que se limpió una mesa
---- Fin algoritmo ----
```

II. Problema en la solución planteada

Un posible problema encontrado se encuentra en los cocineros, ya que uno de ellos puede quedar muy sobrecargado y ser el que prepare la gran mayoría de las comidas, ya que no hay una forma de que el trabajo se distribuya de manera equitativa.

Otro problema surge entre el cliente y el camarero. Cuando un cliente realiza un pedido, luego queda en la espera de dicho pedido que será entregado por el camarero. Sin embargo, en el transcurso desde el pedido hacia la espera del pedido, puede que otro cliente también pida y llegue antes a la instrucción que realiza la espera, con lo cual, el cliente que pidió segundo termina recibiendo la comida antes que el cliente que pidió la comida primero.

III. Comida rápida con cola de mensajes

Solución

Para realizar la solución de este problema, se tomó la solución anterior con semáforos y se traspasó el comportamiento hacia una cola de mensajes convirtiendo los distintos semáforos a un tipo de mensaje y se agregó cierta información adicional para contemplar el nuevo problema de distintos menús.

Respecto a la solución anterior, la sincronización entre los clientes y el limpiador no se ve afectada, ya que la nueva cuestión de distintos menús no afecta el hecho de que una mesa se encuentre o no sucia. Por esto, los semáforos utilizados para esta relación-sincronización entre clientes y limpiador fue llevada a la cola de mensaje definiendo dos tipos en reemplazo de los semáforos utilizados, **TIPO MESA SUCIA** y **TIPO MESA LIMPIA**, los cuales reemplazan a *mesaSucia* y *mesaLimpia* respectivamente.

Cuando un cliente llega al restaurante, tiene que esperar a que haya una mesa limpia (se espera a que haya un mensaje en la cola de mensajes cuyo tipo sea **TIPO MESA LIMPIA**) y una vez que termina de comer y se va notifica que dejó la mesa sucia (se envía un mensaje a la cola de mensajes de **TIPO MESA SUCIA**).

Por otra parte, el limpiador estará siempre atento esperando a que haya una mesa sucia para limpiarla (espera que en la cola de mensajes haya un mensaje cuyo tipo sea **TIPO MESA SUCIA**) y una vez que la limpió notifica que hay una nueva mesa limpia para que otro cliente la utilice (**TIPO MESA LIMPIA**).

En cuanto a la interacción entre el cliente y el camarero se producen cambios en sus algoritmos de coordinación.

En este caso, cuando un cliente realiza un pedido, no alcanza con solo reemplazar el semáforo **comidaPedida** con un tipo que se le corresponda en la cola de mensajes, ya que ahora el cliente tiene que indicar que tipo de menú es el que quiere pedir. Para solucionar esto, además de agregar un tipo de mensajes denominado **TIPO COMIDA PEDIDA** a modo de equivalente al semáforo, se construyó una estructura de mensaje tal que se puede enviar además del tipo de mensaje un argumento adicional que se corresponderá con el menú deseado, entonces, al momento en que el cliente quiere realizar el pedido, envía a la cola de mensajes un mensaje de tipo **TIPO COMIDA PEDIDA** junto con el menú elegido.

Luego, una vez realizada esta acción, el cliente estará esperando a que su menú sea entregado, pero nuevamente el reemplazo directo de semáforo a un nuevo tipo no es suficiente, ya que en el algoritmo anterior el cliente quedaba esperando su pedido, pero ahora tiene que estar esperando que le entreguen únicamente el tipo de menú que solicitó (no puede pedir un menú vegetariano y recibir un menú de carne). Para solucionar este problema, se “fragmentó” el semáforo **comidaEntregada** en dos diferentes tipos para la cola de mensajes, **TIPO COMIDA ENTREGADA CARNE** y **TIPO COMIDA ENTREGADA VEGETARIANA**, lo cual le va a permitir al cliente esperar, en base al menú que pidió, la entrega del menú que le corresponde (si pidió menú carne, espera a que en la cola de mensajes haya un mensaje con tipo TIPO COMIDA ENTREGADA CARNE, si pidió menú vegetariano, espera a que en la cola de mensajes haya un mensaje de tipo TIPO COMIDA ENTREGADA VEGETARIANA).

Desde el lado del camarero también se producen cambios al momento de interactuar con el cliente. Previamente, el camarero esperaba a que un cliente realice un pedido y, cuando esto ocurría, buscaba una comida en la cola de comida para llevar luego al cliente. Sin embargo, ahora que se presentan variaciones de menús, tiene que poder identificar esto. Para solucionar este cambio, al momento en que detecta que alguien realizó un pedido (logra obtener de la cola de mensajes un mensaje con tipo **TIPO COMIDA PEDIDA**), previo a buscar una comida en la cola de comidas, controla el parámetro adicional recibido (que es el tipo de menú solicitado, carne o vegetariano) y en base a este parámetro determina cual es el tipo de comida que tiene que buscar en la cola de comidas.

Luego de que identificó el tipo de menú pedido y de obtener la comida de la cola de comida, tiene que llevárselo al cliente. Aquí nuevamente surgen cambios. En el algoritmo previo, solo se realizaba una notificación mediante el semáforo comidaEntregada, pero ahora se debe diferenciar cual es el tipo de comida que se está entregando para que el cliente que pidió logre recibirla. Para esto, se hace uso de la “fragmentación” previamente mencionada, de modo que si el pedido que detectó era un menú carne, entonces notifica que entrega un menú de carne (envía un mensaje de tipo TIPO COMIDA ENTREGADA CARNE) y si el

pedido que detectó era un menú vegetariano, notifica que entrega un menú vegetariano (enviando un mensaje a la cola de mensajes de tipo TIPO COMIDA ENTREGADA VEGETARIANO).

Finalmente, en cuanto a la sincronización entre el camarero y los cocineros también se realizaron cambios aplicando una estrategia similar a la de clientes-camarero.

Cuando un camarero va a buscar una comida en la cola de comidas, a diferencia de antes que simplemente tomaba una, ahora tiene que ser capaz de sacar una comida según el menú que necesite. La solución a esto se efectuó mediante la “fragmentación” del semáforo **colaDeComidaLlena** previamente utilizado en dos tipos diferentes: **TIPO COLA DE COMIDA LLENA CARNE** y **TIPO COLA DE COMIDA LLENA VEGETARIANO**. Al momento de buscar una comida, como el camarero ya pudo determinar cuál es el tipo de menú requerido, entonces en base a ese tipo busca la comida que necesite, esperando por un mensaje de la cola de mensajes de **TIPO COLA DE COMIDA LLENA CARNE** ó **TIPO COLA DE COMIDA LLENA VEGETARIANO** según requiera.

Luego, también se encuentra un cambio cuando el camarero realiza la notificación de que liberó un espacio en la cola de comidas. Nuevamente se realiza el traspaso del semáforo **colaDeComidaVacía** (lleva control del espacio libre) hacia un nuevo tipo en la cola de mensajes, denominado **TIPO COLA DE COMIDA VACÍA**. Sin embargo, este único cambio no es suficiente para alcanzar el cambio de diseño planteado, ya que la comida sacada puede variar entre diferentes tipos y los cocineros no sabrían a que tipo de menú representaba el espacio liberado. Para ello, se hace uso de la estructura de mensajes que la cola de mensajes representa (en donde se puede especificar el tipo de menú) y el camarero notifica que sacó una comida agregando además la información adicional de cuál fue el tipo de menú liberado.

En cuanto a los cocineros, estos también presentan cambios. Antes, simplemente esperaban a que se libere un espacio en la cola de comidas y preparaban una nueva comida para llenar ese hueco. Ahora, tienen que preparar solo del tipo de comida que se liberó. Para este cambio, se hace uso de la contraparte explicada en la sección anterior de los cambios producidos en el camarero. Al momento de detectar que se liberó un espacio en la cola de comidas (se detecta un mensaje en la cola de mensajes cuyo tipo es **TIPO COLA DE COMIDA VACÍA**), se observa el parámetro adicional que es el que determina el tipo de menú (carne o vegetariano) y en base al menú prepara el necesario y notifica que relleno el espacio liberado, enviando un mensaje a la cola de mensajes de tipo TIPO COLA DE COMIDA LLENA CARNE ó TIPO COLA DE COMIDA LLENA VEGETARIANO según corresponda.

Algoritmos

CLIENTE:

```
----- Inicio algoritmo -----  
-- iterar  
  -- espero una mesa limpia  
  -- miMenu <- selecciono uno de los dos posibles menús  
  -- si miMenu es carne  
    -- notifico que quiero un menú carne  
    -- espero a que llegue el menú carne  
  -- sino  
    -- notifico que quiero un menú vegetariano  
    -- espero a que llegue el menú vegetariano  
  -- comer  
  -- notifico que dejé una mesa sucia  
----- Fin algoritmo -----
```

COCINERO:

```
----- Inicio algoritmo -----  
-- iterar  
  -- espero a que se libere un espacio en una de las colas de comida  
  -- si el espacio liberado es del tipo de menú carne  
    -- preparo un menú carne  
    -- notifico que preparé un menú carne  
  -- sino, (era un tipo de menú vegetariano)  
    -- preparo un menú vegetariano  
    -- notifico que preparé un menú carne  
----- Fin algoritmo -----
```

CAMARERO:

```
----- Inicio algoritmo -----  
-- iterar  
  -- espero a que haya un pedido de comida  
  -- si el pedido es un menú carne  
    -- espero a que en la cola de comidas de carne haya una comida preparada  
    -- notifico que saqué una comida de la cola de comida de carne  
    -- entrego al cliente un menú de carne  
  -- sino, (era un menú vegetariano)  
    -- espero a que en la cola de comidas vegetarianas haya una comida preparada  
    -- notifico que saqué una comida de la cola de comida vegetariana
```



```
-- entrego al cliente un menú vegetariano
---- Fin algoritmo ----
```

LIMPIADOR:

```
---- Inicio algoritmo ----
-- iterar
  -- espero a que haya una mesa sucia
  -- limpio la mesa
  -- notifico que hay una nueva mesa limpia
  ---- Fin algoritmo ----
```

2. Problemas

1. Lectura

Breve Introducción a la Arquitectura de Android

El sistema operativo Android es una pila de componentes de software donde esta se divide en cuatro capas diferentes. Como primera capa se encuentra el Kernel, luego las librerías nativas, después los frameworks de aplicación y por último están las aplicaciones.

a)

1. Kernel :

Esta primera capa proporciona las funcionalidades básicas del sistema operativo tales como la gestión de procesos, memoria, y dispositivos (camara, pantalla, teclado, etc). El kernel utilizado en el sistema operativo Android se basa en el kernel de la serie Linux debido a que este es realmente bueno en operaciones básicas como la networking, además de contar con una gran variedad de controladores de dispositivos , y se encuentra modificado para cumplir con las necesidades de una mejor gestión de memoria, procesos y manejo de energía además contar con un mecanismo especial de comunicación entre procesos para acoplarse de una manera más adecuada a los recursos limitados del sistema. Se aplican además otro conjunto de modificaciones para el SO Android tales como:

a. *Dispositivo de alarma:*

El kernel de android lleva a cabo un conjunto de operaciones y planificación que permiten ejecutar aplicaciones cuando ocurre un determinado evento

b. *Binder:*

Como en los dispositivos móviles se cuenta con recursos limitados en comparación con las pc ,esto requiere que la comunicación entre procesos (IPC) sea eficiente en términos de velocidad y gestión de memoria, por lo tanto Android introduce el Binder.

c. *Manejo de energía:*

Se agregó una gestión de energía propia para poder adaptarse según las necesidades de los dispositivos móviles.

d. *Low Memory killer:*

Esto entra en función cuando el dispositivo se queda sin memoria , de manera que se encarga de elegir qué proceso matar.A

la hora de elegir qué proceso es seleccionado se define por la política definida por el propio dispositivo del usuario.

e. *Kernel Debugger:*

Como el sistema Operativo Android es de código abierto y además el kernel de linux también lo es, esto significa que cualquiera puede realizar cambios lo que puede llevar a que estos no funcionen de manera adecuada por lo tanto es necesario contar con un depurador de kernel.

f. *Logger:*

El logger es importante ya que se encarga de registrar todos los mensajes del sistema para la resolución de problemas.

g. *Ashmem:*

La memoria compartida de android, es otro de los componentes del kernel el cual facilita el uso compartido y conservación de la memoria para lograr un mayor soporte a dispositivos con poca memoria.

2. Librerías Nativas:

Por encima de la capa del Kernel se encuentran las librerías nativas de Android , la cual permite manejar diferentes tipos de datos específicos del hardware. Esta capa se divide en dos partes:

Por un lado se encuentran las librerías de Android las cuales se encargan de las tareas pesadas, además de proporcionar performance detrás de la plataforma android. Entre las librerías disponibles se encuentran por ejemplo SQLite (base de datos relacional), Framework de medios (permite grabar y reproducir numerosos formatos de audio, video e imágenes), WebKit (una herramienta que permite mostrar contenido HTML) entre otras.

La segunda parte es el tiempo de ejecución de Android denominada Dalvik Virtual machine (DVM) y las librerías centrales de java. Esta máquina virtual está diseñada para pequeños sistemas los cuales cuentan con un tamaño de ram reducido y una cpu lenta. La ventaja de contar con DVM es la consistencia en los tiempos de ejecución y portabilidad que provee a las aplicaciones.

3. Framework de aplicación:

Esta capa se encuentra arriba de las librerías nativas. Provee una importante interfaz de programación de aplicaciones y distintos servicios de nivel superior en forma de clases de Java. La API se encuentran abiertas para todos los desarrolladores y de esta manera puedan acceder a todo el framework de las API para los programas principales que simplifica la

reutilización de los componentes de las mismas. Existen diferentes componentes para las aplicaciones, cada uno de ellos con diferente ciclo de vida y un propósito, el cual describe cómo se crea y destruye el componente. Entre los distintos componentes se encuentran las actividades que representan una única ventana con interfaz de usuario. En estas aplicaciones, las actividades funcionan juntas para lograr una experiencia de usuario coherente. En las aplicaciones de múltiples actividades, se establece una actividad principal que se presenta al usuario al iniciar la misma, que cuando realizan distintas acciones cada actividad puede iniciar otra actividad, donde la anterior se detiene y se guarda el estado en una pila para su posterior uso. Además, para la gestión de estas actividades se cuenta con un gestor de actividades el cual se encarga del ciclo de vida de las aplicaciones.

Por otra parte, en esta capa se encuentran otros componentes denominados servicios los cuales se ejecutan en segundo plano sin proporcionar una interfaz de usuario y se encargan de realizar las operaciones de larga duración. Otro de los componentes que se encuentra en la capa de framework de aplicación son los proveedores de contenido que proporcionan los datos compartidos entre aplicaciones además de la gestión del acceso a los datos de otras aplicaciones.

Por último en esta capa se encuentra el package manager, window manager, hardware services, servicio de telefonía y servicio de ubicación.

4. Aplicaciones:

Esta es la capa que se encuentra en lo más alto de la arquitectura de Android. En ella se encuentran todas las aplicaciones nativas que vienen preinstaladas así como también las que son instaladas por el usuario. Esta capa es con la cual interactúa el usuario promedio de android. Además un desarrollador puede escribir su propia aplicación y reemplazarla con la aplicación existente.

b)

Elementos representativos para este tipo de sistemas operativos:

- La gestión de recursos en este tipo de sistemas operativos es de vital importancia, ya que estos dispositivos por lo general cuentan con recursos limitados como la cantidad de RAM y procesadores lentos con respecto a los de un ordenador este requerimiento de optimización se cumple con la utilización de Dalvik Virtual machine. Por otra parte, la utilización de Zygoter ayuda a compartir código en la VM Dalvik logrando un menor tiempo de inicio mínimo.
- Estos sistemas operativos no son reconocidos principalmente por su seguridad, ya que estos cuentan con distintos problemas como es

darle acceso irrelevante a las aplicaciones durante la instalación de aplicaciones, por lo tanto después de la instalación estas obtienen acceso a nuestros datos personales. Otro de los problemas de seguridad es la espera que se debe tener para la actualizaciones de seguridad.

- Estos sistemas operativos tienen el enfoque a dispositivos móviles. Un requisito importante de estos dispositivos es contar con servicios de telefonía los cuales proveen la capacidad de enviar o recibir mensajes o llamadas y la conectividad.

c)

El artículo cumple con la intención propuesta por el autor de poder dar a conocer un pantallazo detrás de la arquitectura,, las diferentes características y el funcionamiento del sistema operativo Android además de poder visualizar todos los cambios que se dieron sobre el kernel de linux para su optimización en dispositivos con recursos limitados como Smartphones o tablets.

Consideramos que con la descripción de la arquitectura del sistema operativo android podemos hacer una relación con los conceptos previamente vistos en la materia como lo son los sistemas operativos, el manejo de memoria, procesos , planificación y sincronización de procesos. Uno de los aspectos más interesantes son las mejoras que se agregaron al kernel de linux respecto a la intercomunicación entre procesos (IPC), la cual permite que los procesos intercambien datos entre sí, donde Android introduce Binder para la eficiencia en términos de velocidad y memoria. De esta manera se obtiene una mejora de rendimiento a través de la memoria compartida, donde no hay replicación de objetos sino que estos se pasan por referencia logrando así una mayor eficiencia en término de memoria además de no tener sobrecarga de procesamiento.

El artículo además introduce los problemas más frecuentes que suceden en Android donde nos llamó la atención el problema con el sistema de permisos que cada vez que instalamos una aplicación estamos concediendo permisos dando así el acceso a nuestros datos personales.

2. Problemas conceptuales

1. Copy-on-write

En este inciso se tienen dos procesos, A y B, donde B invoca fork y el kernel utiliza la estrategia copy-on-write para la implementación del mismo. Al invocar el fork() se crea un proceso hijo como duplicado de su padre. Generalmente fork() trabaja creando una copia del espacio de direcciones del padre para el hijo, duplicando las páginas pertenecientes al padre pero como se utiliza la estrategia copy-on-write los procesos padres e hijo comparten inicialmente las mismas páginas, por lo tanto cuando el proceso B invoca el fork(), el proceso hijo de B va a compartir las páginas con B.

a)

Diagrama de asignación:

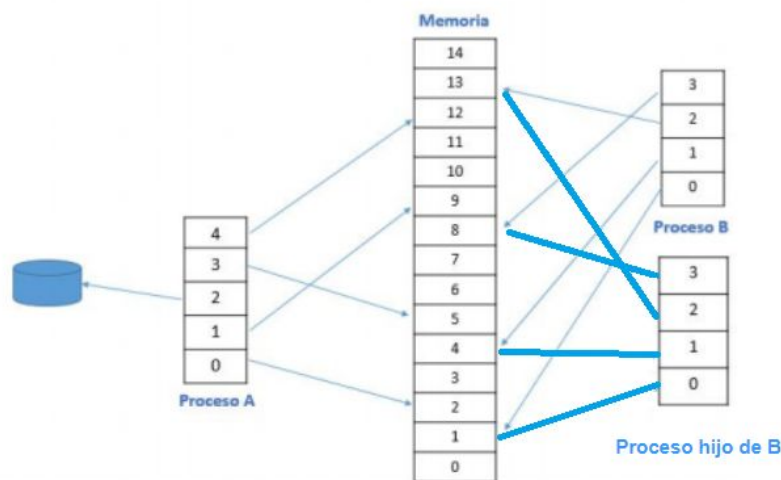
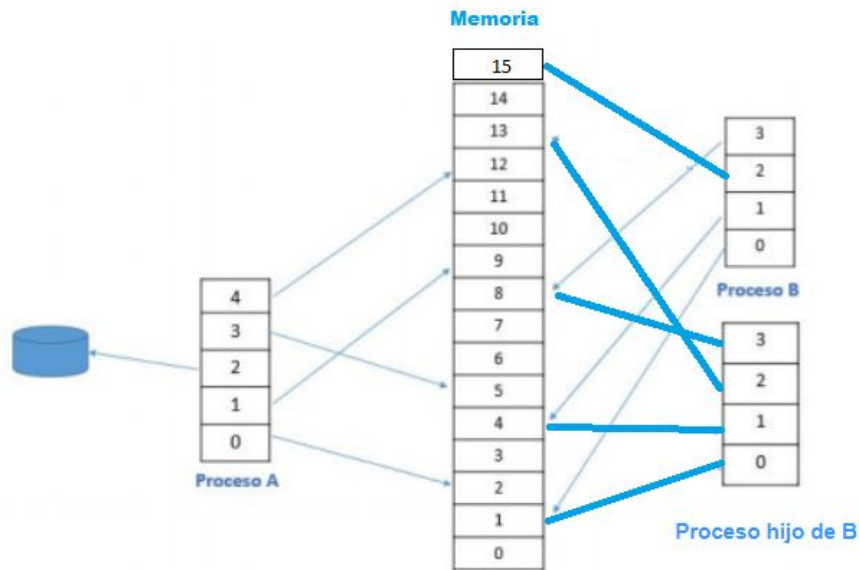


Diagrama de asignación después de que B modifique la página 2:



Luego B deja de compartir la página 2 con el proceso hijo y las páginas no modificadas serán compartidas entre el proceso B y el hijo.

b) Tabla de páginas para el padre y el hijo:

Antes de la modificación de la página 2

| Proceso B | | | | |
|-----------|--------|---------------|------------------|----------------------|
| Frame | Página | Bit de Valido | Bit de escritura | Bit de copy-on-write |
| 8 | 3 | 1 | 0 | 1 |
| 13 | 2 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |

| Proceso hijo de B | | | | |
|-------------------|--------|---------------|------------------|----------------------|
| Frame | Página | Bit de Valido | Bit de escritura | Bit de copy-on-write |
| 8 | 3 | 1 | 0 | 1 |
| 13 | 2 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |

Después de la Modificación de la página 2

| Proceso B | | | | |
|---------------------|---------------|----------------------|-------------------------|-----------------------------|
| Frame | Página | Bit de Valido | Bit de escritura | Bit de copy-on-write |
| 8 | 3 | 1 | 0 | 1 |
| Copia de 13 (15) | 2 | 1 | 1 | 0 |
| 4 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |

| Proceso hijo de B | | | | |
|--------------------------|---------------|----------------------|-------------------------|-----------------------------|
| Frame | Página | Bit de Valido | Bit de escritura | Bit de copy-on-write |
| 8 | 3 | 1 | 0 | 1 |
| 13 | 2 | 1 | 0 | 0 |
| 4 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |

2. FAT

a)

| Nombre | tipo | fecha | Nro.Bloque |
|-----------------|------|------------|------------|
| Actividades.txt | F | 29-09-2020 | 3 |
| Act-Labo1.txt | F | 10-10-2020 | 1 |
| Act-Labo2.txt | F | 15-10-2020 | 2 |
| Prueba | D | 15-10-2020 | 8 |
| Informe | F | 18-10-2020 | 9 |

| | | | |
|----|----|----|--|
| 1 | * | 16 | |
| 2 | 4 | 17 | |
| 3 | 15 | 18 | |
| 4 | 5 | 19 | |
| 5 | * | 20 | |
| 6 | 7 | 21 | |
| 7 | * | 22 | |
| 8 | * | 23 | |
| 9 | 10 | 24 | |
| 10 | 11 | 25 | |
| 11 | 12 | 26 | |
| 12 | * | 27 | |
| 13 | | 28 | |
| 14 | | 29 | |
| 15 | 6 | 30 | |

b) Mapa de Bits

Lista de espacio libre implementada como un mapa de bits para la gestión del espacio libre, cada bloque está representado por 1 bit. Considerando que si el bloque se encuentra libre, el bit será igual a 1, caso contrario el bit será igual a 0. La sucesión de bits que contendrá respecto a los bloques dados es:

0000000000000110111111111111

El mapa de bits ocuparía un tal de 30 bits, siendo un bit por cada bloque respectivamente.

c) Lista Enlazada

La lista enlazada consiste en enlazar todos los bloques de disco libres, manteniendo un puntero al primer bloque libre. El primer bloque contendrá un puntero al siguiente bloque libre del disco.

Por lo tanto si se utiliza una lista enlazada para la gestión del espacio libre requerirá un puntero al primer bloque libre (en este caso al bloque número 13) por lo tanto requerirá 4 bytes.

Si se pierde el primer puntero, el sistema Operativo podrá reconstruir la lista de espacio libre mediante la tabla de archivos. De esta manera se obtendrá los bloques ocupados y se podrá obtener el primer bloque libre, de esta manera la lista enlazada se reconstruye.

3. Tablas de información

