INFORME: PROYECTO SISTEMAS OPERATIVOS

Comisión 41: Jano Axel Lockhart, Bautista Graff Bohn

Sistemas Operativos
Universidad Nacional del Sur
2022

ÍNDICE

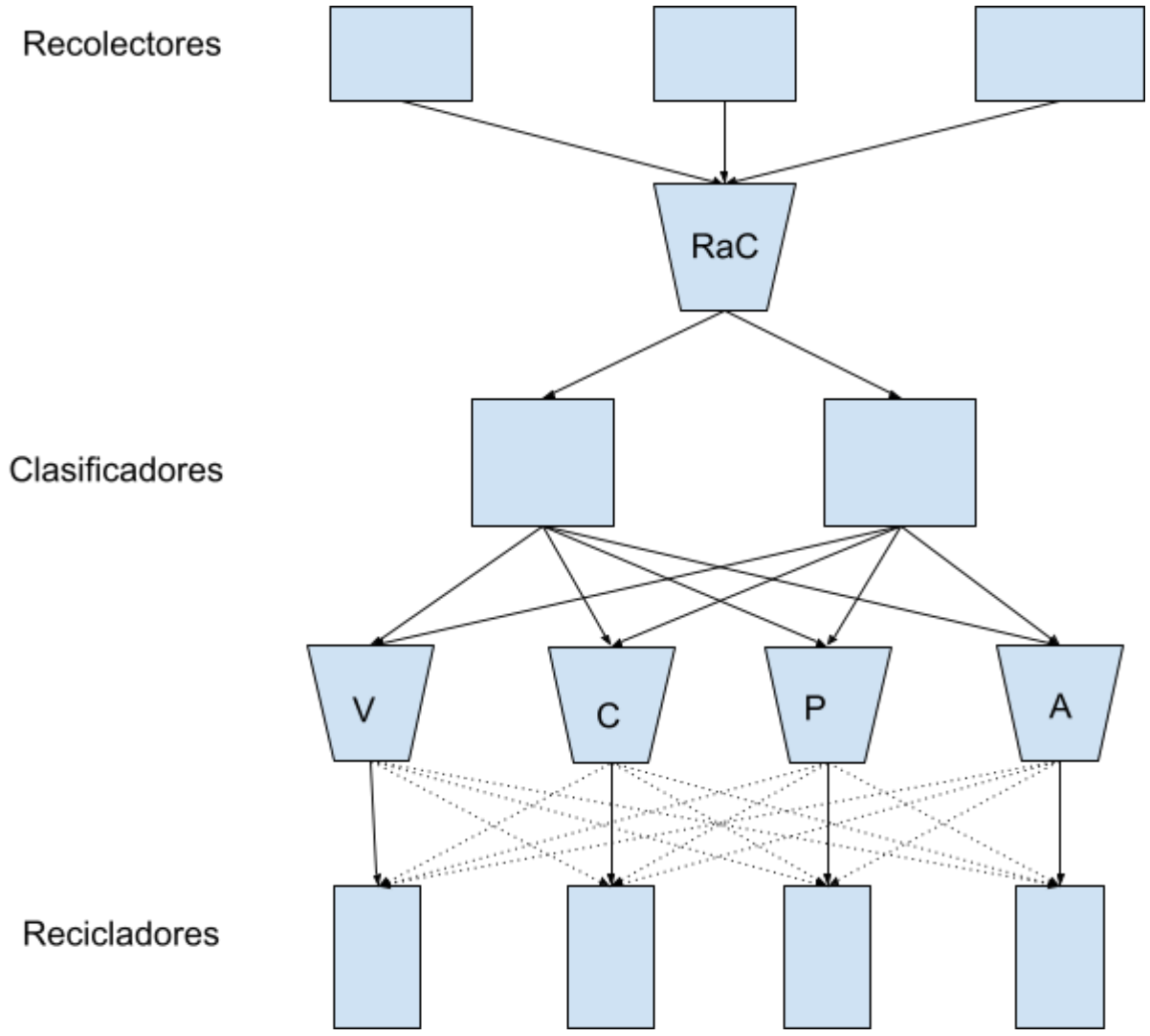
SECCIÓN 1

1.1 Procesos, threads y Comunicación

1.Planta de Reciclado

Inciso a

En este inciso, un proceso principal primero crea todos los pipes. Luego crea los procesos hijos recolectores, clasificadores y recicladores. Antes de iniciar su tarea, cada proceso se encarga de cerrar las entradas de los pipes que no utilizará.



En la imagen, los procesos son rectángulos y los pipes son los trapecios. Las flechas sólidas representan la basura que recibe cada proceso y las flechas punteadas apuntando a los recicladores indican que ese proceso puede tomar basura de otro pipe para ayudar.

Para la implementación de este inciso utilizamos 5 pipes. Utilizamos el pipe RaC que es la comunicación entre los recolectores y clasificadores; y los pipes restantes

simbolizan cada uno de los materiales que son clasificados y serán reciclados. Estos son, vidrio (V), cartón ©, plástico (P) y aluminio (A). Como los recicladores deben poder ayudar a otros, estos cuatro pipes tienen el lado de lectura como no bloqueante.

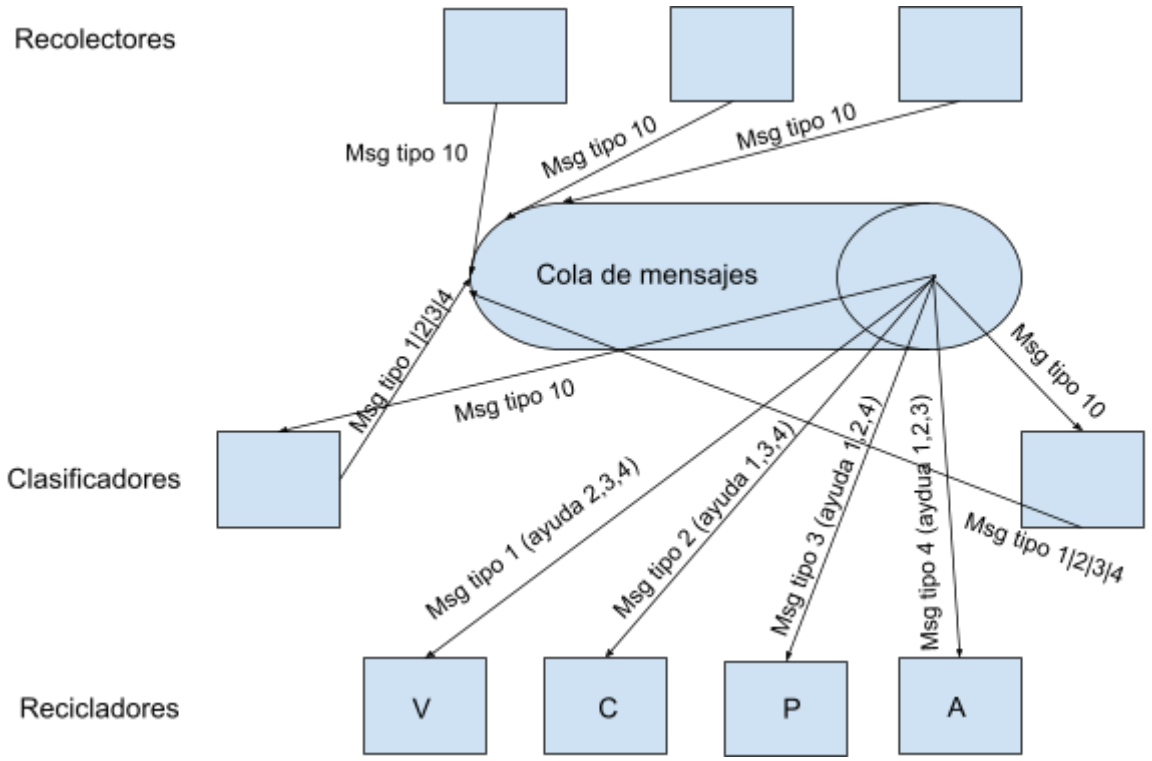
Los recolectores generan uno de los cuatro tipos de basura aleatoriamente y la envían al pipe RaC.
Luego, los clasificadores reciben basura de ese pipe, la clasifican y la envían al pipe correspondiente.
Por último, los recicladores revisan su pipe correspondiente para ver si tienen basura para reciclar. Si este es el caso, reciclan la basura. En caso contrario, van consultando uno por uno al resto de pipes para reciclar otro tipo de basura para ayudar. Si no encuentra basura en el resto de pipes, el reciclador toma mate por un tiempo determinado en el cual no reciclará.

En la implementación de los distintos procesos se decidió colocar sleeps luego de que cada uno realice su acción para permitir una mejor lectura de los mensajes mostrados por la consola.

Compilación y Ejecución:
Para compilar los archivos se debe ejecutar plantaPipes.sh. Luego, para poner en funcionamiento la planta se deberá ejecutar el archivo plantaReciclado.

Inciso b

Para este inciso, el proceso principal crea todos los procesos y la cola de mensajes. Luego, cada proceso se vincula a esta última y realizan su tarea.



En la imagen los procesos son cuadrados, las flechas representan los mensajes y el cilindro es la cola de mensajes.

Para esta implementación utilizamos una única cola de mensajes. Los mensajes de tipo 10 son los de comunicación entre Recolectores y Clasificadores, mientras que los mensajes de tipo 1 son los de Clasificadores a Reciclador de vidrio, los de tipo 2 de Clasificadores a Reciclador de cartón, los de tipo 3 de Clasificadores a Reciclador de plástico y los de tipo 4 de Clasificadores a Reciclador de aluminio.

En un primer momento, los recolectores generan uno de los cuatro tipos de basura aleatoriamente y lo envían a la cola de mensajes con el tipo de mensaje 10.
Luego, los clasificadores leen los mensajes de tipo 10 de la cola y según el tipo de basura lo clasifica y le asigna el tipo de mensaje correspondiente. Para el vidrio es el 1, para el cartón es 2, plástico es 3 y aluminio es 4. Este mensaje es colocado en la cola de mensajes.
Por último, los recicladores realizan una lectura no bloqueante sobre el tipo de mensaje que le corresponde. Si lo encuentran, reciclan la basura. En caso de no encontrar ninguno, intentan leer el resto de tipos de mensajes de basura utilizando un número negativo en el tipo de mensaje de msgrcv. Si lo encuentran, el reciclador anuncia que ayudó a otro reciclador. En caso de que justo haya entrado a la cola un mensaje de su tipo de basura, es decir, que se ayudaría a sí mismo, muestra el mismo mensaje que si lo hubiese reciclado.
Si se da el caso de que los recicladores no encuentran mensajes de basura ni de su tipo ni de otro tipo para reciclar, toman mate por un tiempo determinado.

En la implementación de los distintos procesos se decidió colocar sleeps luego de que cada uno realice su acción para permitir una mejor lectura de los mensajes mostrados por la consola.

IMPORTANTE: si la ejecución es abortada se deberá eliminar la cola de mensajes utilizando el comando ipcrm para que los mensajes que hayan quedado no generen problemas con ejecuciones posteriores.

Compilación y Ejecución:
Para compilar los archivos se debe ejecutar plantaColaMensajes.sh. Luego, para poner en funcionamiento la planta se deberá ejecutar el archivo plantaColaMensajes.

2.Minishell

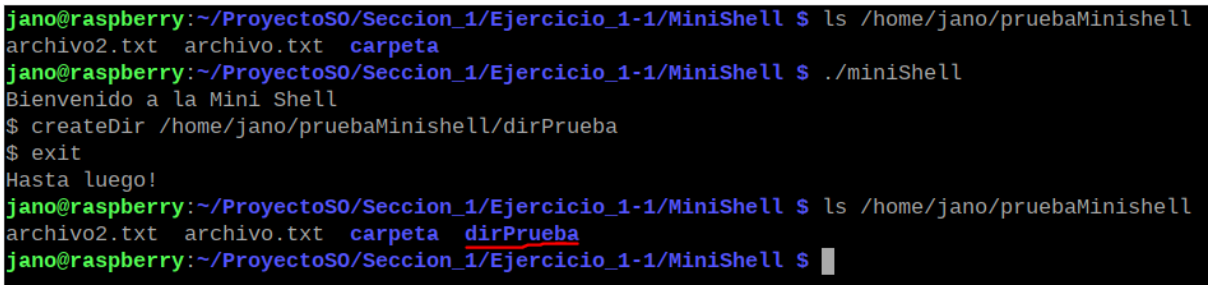
Para la implementación, el proceso principal se encarga de leer los comandos que el usuario ingresa por consola y separa cada uno de los argumentos ingresados. Luego, crea un nuevo proceso y se le carga la imagen correspondiente al comando ingresado, enviándole los argumentos como parámetros. Si el comando no existe o es ingresado con un nombre, parámetros o cantidad de parámetros incorrecta, se mostrará el mensaje de error correspondiente.

Al ingresar un comando, su longitud no debe superar los 512 caracteres.

Inciso a: Crear un directorio

createDir [nombre directorio]:
Crea un directorio con nombre "nombre directorio". Si se quiere crear un directorio en una ubicación distinta de donde está la minishell, deberá especificar la ruta completa en “nombre directorio”.

En esta imagen se creó un directorio con nombre “dirPrueba” en la misma ubicación donde está la minishell.



En esta imágen se creó un directorio con nombre “dirPrueba” en una ubicación distinta a la que se encuentra la minishell

Inciso b: Eliminar un directorio

deleteDir [nombre directorio]

Borra el directorio con nombre "nombre directorio". Si se quiere borrar un directorio que se encuentra en otra carpeta en la que no está la minishell, deberá especificar la ruta completa en “nombre directorio” de la misma forma que al crear un directorio.

```
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls
archivoPrueba  createDir.c   deleteDir    help      listDir.c    miniShell.sh  setFilePer.c  viewFile.c
ayuda.txt      createFile    deleteDir.c  help.c    miniShell    per.txt       utils
createDir      createFile.c  dirPrueba    listDir   miniShell.c  setFilePer    viewFile
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ./miniShell
Bienvenido a la Mini Shell
$ deleteDir dirPrueba
$ exit
Hasta luego!
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls
archivoPrueba  createDir.c   deleteDir    help.c    miniShell    per.txt       utils
ayuda.txt      createFile    deleteDir.c  listDir   miniShell.c  setFilePer    viewFile
createDir      createFile.c  help         listDir.c miniShell.sh setFilePer.c  viewFile.c
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $
```

En esta imágen se eliminó el directorio de nombre “dirPrueba”

```
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell/
archivo2.txt  archivo.txt  archPrueba  carpeta  dirPrueba
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ./miniShell
Bienvenido a la Mini Shell
$ deleteDir /home/jano/pruebaMinishell/dirPrueba
$ exit
Hasta luego!
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell/
archivo2.txt  archivo.txt  archPrueba  carpeta
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $
```

Aquí se eliminó el directorio dirPrueba de una ubicación distinta a la de la minishell

Inciso c: Crear un archivo

createFile [nombre archivo]

Crea un archivo vacío con el nombre "nombre archivo". Si se quiere crear un archivo en otro directorio en el que no está la minishell, deberá especificar la ruta completa en “nombre directorio”.

```
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell
archivo2.txt  archivo.txt  carpeta  dirPrueba
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ./miniShell
Bienvenido a la Mini Shell
$ createFile /home/jano/pruebaMinishell/archPrueba
$ exit
Hasta luego!
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell
archivo2.txt  archivo.txt  archPrueba  carpeta  dirPrueba
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $
```

En el ejemplo se creó un archivo denominado “archPrueba”.

Inciso d: Listar contenido de un directorio

listDir [nombre directorio]

Muestra todos los elementos que contiene el directorio con nombre "nombre directorio". Si se quiere listar el contenido de un directorio en una ubicación distinta al de la minishell, se debe especificar su ruta completa en “nombre directorio”.

```
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell
archivo2.txt  archivo.txt  archPrueba  carpeta  dirPrueba
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ./miniShell
Bienvenido a la Mini Shell
$ listDir /home/jano/pruebaMinishell
..
archivo2.txt
archPrueba
carpeta
dirPrueba
archivo.txt
.
$ exit
Hasta luego!
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $
```

En el ejemplo se listaron los contenidos de un directorio que no está con la minishell

Inciso e: Mostrar el contenido de un archivo

viewFile [nombre archivo]

Muestra el contenido de un archivo con nombre "nombre archivo". Si el archivo no se encuentra en la misma ubicación que la minishell, se deberá especificar la ruta completa en “nombre archivo”.

```
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ls /home/jano/pruebaMinishell
archivo2.txt  archivo.txt  archPrueba  carpeta  dirPrueba
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ cat /home/jano/pruebaMinishell/archPrueba
TEXTO PARA MOSTRAR

TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR
TEXTO PARA MOSTRAR
TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR

TEXTO PARA MOSTRAR
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $ ./miniShell
Bienvenido a la Mini Shell
$ viewFile /home/jano/pruebaMinishell/archPrueba
TEXTO PARA MOSTRAR

TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR
TEXTO PARA MOSTRAR
TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR TEXTO PARA MOSTRAR

TEXTO PARA MOSTRAR
$ exit
Hasta luego!
jano@raspberry:~/ProyectoSO/Seccion_1/Ejercicio_1-1/MiniShell $
```

En este ejemplo se muestra el contenido del archivo archPrueba

Inciso f: Mostrar una ayuda con los comandos posibles

help

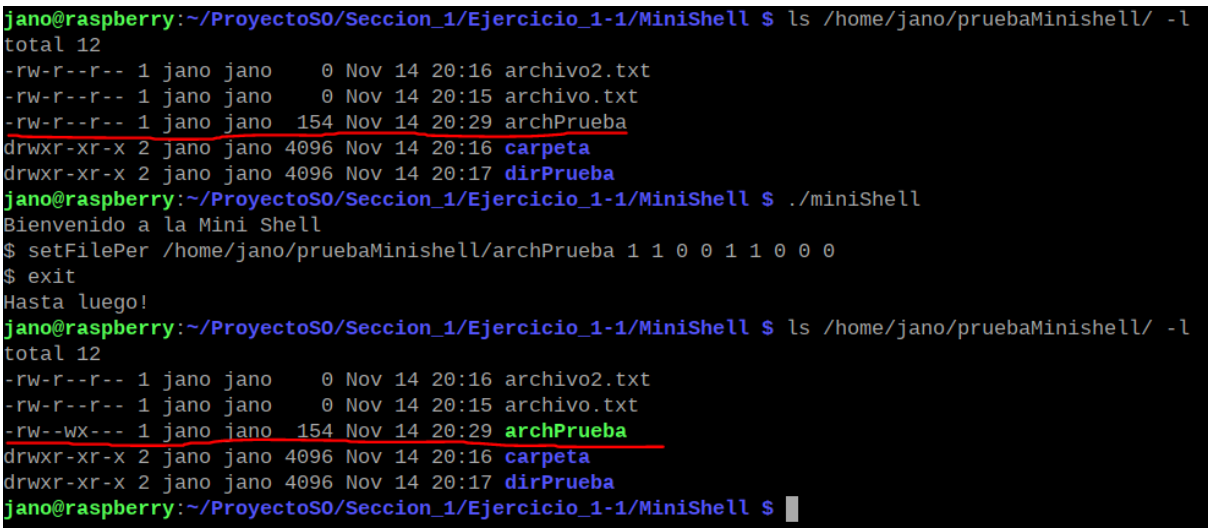
Muestra una ayuda dónde indica las funcionalidades y el uso de la minishell

Inciso g: Modificar los permisos de un archivo

setFilePer [ubicación archivo] [dueño read] [dueño write] [dueño execute] [grupo read] [grupo write] [grupo execute] [otros read] [otros write] [otros execute]

Establece los permisos de un archivo en “ubicación archivo” según los argumentos ingresados.

Los permisos read, write, execute pueden ser 0 o 1. 0 para no otorgarlo, 1 para otorgarlo. Se debe establecer el valor de todos los permisos en una llamada al comando setFilePer, es decir que la cantidad de 1 's sumada a la de 0’s siempre debe ser nueve.



Aquí se asignaron permisos sobre “archPrueba” para que el dueño (usuario) pueda leerlo y escribirlo, y que el grupo pueda escribirlo y ejecutarlo.
El resto de permisos no fueron otorgados.

Compilación y Ejecución:
Para compilar los archivos y ejecutar la minishell se debe ejecutar miniShell.sh.
Si se quiere iniciar nuevamente la minishell, sin volver a compilar nuevamente, se puede ejecutar el archivo miniShell.

1.2 Sincronización

1.Secuencia

Inciso a: Hilos y Semáforos

Para la implementación de este ejercicio, se revisaron los patrones que aparecen en la secuencia y se armó el siguiente pseudocódigo.

```
rutinaA{
        wait(DoEA)
        wait(FA)
        A
        signal(ABC)
}

rutinaB{
        wait(CB)
        wait(ABC)
        B
        signal(BC)
        signal(BCDoE)
}

rutinaC{
        wait(BC)
        wait(ABC)
        C
        signal(CB)
        signal(BCDoE)
}

rutinaD{
        wait(BCDoE)
        D
        signal(DoEF)
        signal(DoEA)
}

rutinaE{
        wait(BCDoE)
        D
        signal(DoEF)
        signal(DoEA)
}
```

```
rutinaF{
        wait(DoEF)
        wait(DoEF)
        F
        signal(FA)
        signal(FA)
}
```

Los patrones en los que nos basamos fueron los siguientes:
-Después de una A siempre le sigue una B o una C. Por eso A tiene signal(ABC) y, B y C hacen wait sobre ese semáforo.

-La B y la C se intercalan luego de la A. Para esto están los semáforos CB y BC.

-Después de una B o una C siempre le sigue (DoE). Entonces B y C tienen signal(BCDoE) y las rutinas C y E lo esperan. Como debe poder ejecutarse cualquiera de los dos, tanto C como E compiten por el semáforo BCDoE y el primero que lo toma es el que se ejecuta.

-Hay una F después de que se hayan ejecutado dos (DoE). Por eso D y E hacen signal(DoEF) y F tiene dos wait(DoEF).

-Después de una F siempre le sigue una A, pero antes de una A puede haber una F o (DoE). Como siempre se ejecuta A luego de (DoE), excepto cuando hay una F de por medio, D y E hacen signal(DoEA) y A la espera.
Por otra parte, la A aparece dos veces antes de que F vuelva a aparecer, es por esto
que F realiza dos signal(FA) y A solo hace un wait(FA)

Semáforos y su inicialización:
FA= 2
ABC= 0
CB= 1
BC= 0
BCDoE= 0
DoEF= 0
DoEA= 1

Como A es la primera letra que aparece, inicializamos FA=2 (para que no aparezca la F hasta que se haya ejecutado dos A) y DoEA=1.
Como B es lo que sigue a la primera A, inicializamos CB=1
El resto de semáforos se los inicializa en 0

En la implementación hay sleeps con 0 de duración luego de cada letra, por lo que éstas aparecerán instantáneamente por consola. Si se quiere que vayan más lento se debe cambiar la macro TIEMPO_SLEEP por un número mayor a 0.

Compilación y Ejecución:
Para compilar el archivo se debe ejecutar secuenciaHilosSemaf.sh. Luego, para imprimir la secuencia se debe ejecutar secuenciaHilosSemaf.

Inciso b: Procesos y Pipes

En la implementación de este inciso, realizamos una traducción directa del inciso anterior: los semáforos son pipes, los signal son write y los wait son read sobre los pipes correspondientes.

Compilacion y Ejecucion:
Para compilar el archivo se debe ejecutar secuenciaProcesosPipe.sh. Luego, para imprimir la secuencia se debe ejecutar secuenciaProcesosPipe.

2.Puente de una sola Mano

Inciso I

Para la implementación de este inciso, primero creamos el siguiente pseudocódigo para los autos provenientes del sur.

```
AutoSur

//Protocolo de entrada
wait(entrarSur)
if(trywait(cantSur)=error)
        wait(puenteLibre)
else
        signal(cantSur)
signal(cantSur)
printf("Entró al puente")
signal(entrarSur)

//Sección crítica
        printf("Estoy pasando por el puente")

//Protocolo de salida
wait(entrarSur)
wait(cantSur)
printf("Salió del puente")
if(trywait(cantSur)=error)
        signal(puenteLibre)
else
        signal(cantSur)
signal(entrarSur)
```

Con el auto norte es análogo, sólo que en lugar de hacer wait y signal sobre los semáforos entrarSur y cantSur, lo hace sobre entrarNorte y cantNorte respectivamente.

Los semáforos entrarSur y entrarNorte son semáforos binarios (mutex) para asegurar la exclusión mutua de entrada al puente entre los autos del sur y norte respectivamente, ya que al entrar al puente se consulta y modifica el semáforo cantSur o cantNorte.

Los semáforos cantSur y cantNorte son semáforos de conteo que indican la cantidad de autos provenientes del sur y norte que hay en el puente respectivamente.

Semáforo puenteLibre es un semáforo binario que indica si el puente está libre o no. Éste asegura exclusión mutua entre los autos provenientes del norte y del sur.

En este ejercicio los autos están modelados con hilos.
Para simular la llegada de múltiples autos, se crearon dos hilos que se encargan de crear hilos de autos de las direcciones Norte y Sur.

Dentro del código, el semáforo permDir es el semáforo puenteLibre del pseudocódigo, entSalSur es entrarSur, entSalNorte es entrarNorte y cantSur y cantNorte están igual.

Para la implementación se decidió utilizar sleeps para el tiempo de aparición entre dos autos y para el tiempo que tardan en cruzar el puente, para así poder seguir los mensajes que van apareciendo por consola.
Además cada auto tiene un número que lo identifica según la hora en que apareció para poder distinguir qué auto entró o salió del puente.

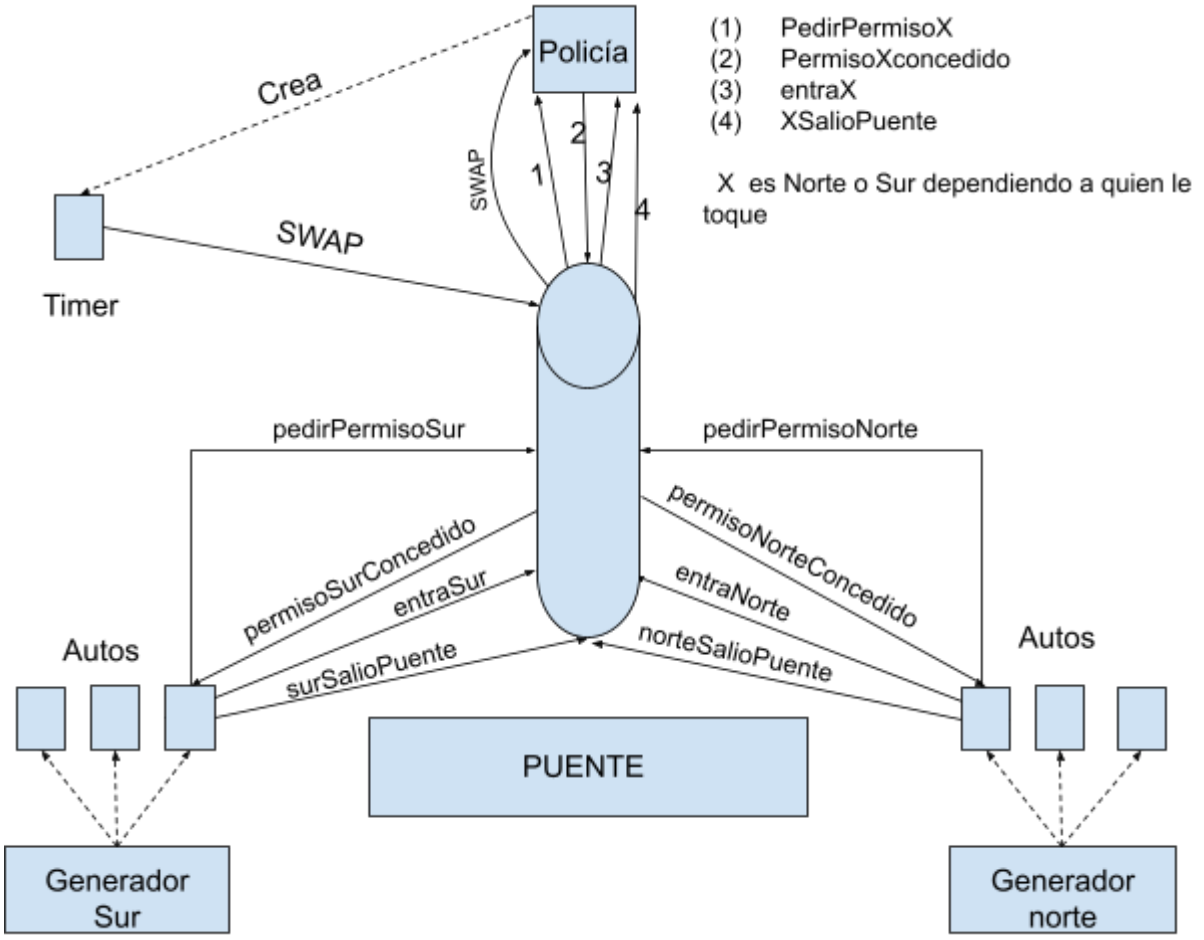
Compilación y Ejecución:
Para compilar el archivo se debe ejecutar puenteHilosSemaf.sh. Para que comiencen a cruzar autos por el puente se debe ejecutar puenteHilosSemaf.

Inciso II

Asumiendo que siempre hay autos en ambas entradas del puente esperando a pasar por él, el problema que presenta la solución planteada es que solo los autos con la misma dirección del que logró entrar primero pueden cruzarlo. Es decir, que si el primer auto en entrar proviene del Norte, solo pasarán por el puente autos del Norte y los del Sur se quedarán esperando indefinidamente, ya siempre habrá un auto del Norte pasando.

Inciso III

Para solucionar el problema explicado en el inciso anterior se optó por tener un proceso árbitro, llamado policía, que se encarga de alternar, cada cierto tiempo, la dirección que tiene paso sobre el puente.



En la imagen, los procesos están representados con rectángulos (a excepción del puente que no es un proceso) y la cola de mensajes es el cilindro. Cada flecha sólida con su etiqueta representa un tipo de mensaje que se envía o recibe. Las flechas punteadas indican que el proceso crea otros procesos.

En la implementación, el proceso principal se encarga de crear el policía y los generadores de sur y norte. Estos últimos se encargan de crear procesos Auto de la dirección correspondiente.

El policía empieza dando prioridad a los autos provenientes del sur, es decir que recibe pedidos de permiso de autos del Sur. Además crea un timer para que le avise cuando debe cambiar la dirección de paso. Los pedidos de permiso de los autos del norte quedan en la cola de mensajes.
Cuando el policía recibe un pedido de permiso, envía un mensaje al auto concediéndoselo (permisoSurConcedido). Luego, el auto anuncia que entra al puente enviando un mensaje (entraSur) para que el policía pueda llevar la cuenta de la cantidad de autos pasando por el puente. Una vez que el auto termina de cruzar el puente, lo anuncia enviando un mensaje (surSalioPuente).
Todo esto se repite hasta que el timer que creó el policía le avisa que debe cambiar de dirección. En este momento, el policía deja de conceder permiso de paso a los autos del sur, lee todos los mensajes de autos que avisaron que salieron del puente y espera por aquellos que todavía se encuentran cruzándose. Una vez que no quedan más autos, cambia la dirección que tiene paso a Norte, es decir, que a partir de ahora recibirá y concederá permiso a autos que provengan de esta dirección.

Crea un nuevo timer y lo explicado anteriormente se vuelve a repetir pero ahora con autos del Norte.

Nótese que cuando un auto aparece y envía su petición de permiso al policía, pero no le toca pasar a los de su dirección, el mensaje queda en la cola de mensajes en el orden en el que fueron emitidos. Entonces, cuando el policía cambie la dirección, recibirá esos mensajes en orden de llegada y concederá los permisos en ese orden. Por lo tanto, en algún momento todo auto que desee atravesar el puente logrará hacerlo, solucionando así el problema de la solución anterior explicada en el inciso II.

En la implementación, se agregó un sleep entre la aparición de autos y el tiempo que tardan en cruzar el puente para que los mensajes puedan seguirse cuando se muestran por consola.

IMPORTANTE: si la ejecución es abortada se deberá eliminar la cola de mensajes utilizando el comando ipcrm para que los mensajes que hayan quedado no generen problemas con ejecuciones posteriores.

Compilación y Ejecución:
Para compilar los archivos se debe ejecutar puente.sh. Para que comiencen a cruzar autos por el puente se debe ejecutar puenteProcesosColaMsg.

SECCIÓN 2

2.1 Lectura

El artículo elegido para esta actividad fue “Allied Telesis AlliedWare Plus Operating System”. Se decidió realizar un podcast para presentar la idea. El audio correspondiente se encuentra adjuntado en la carpeta Seccion_2 junto a este informe.

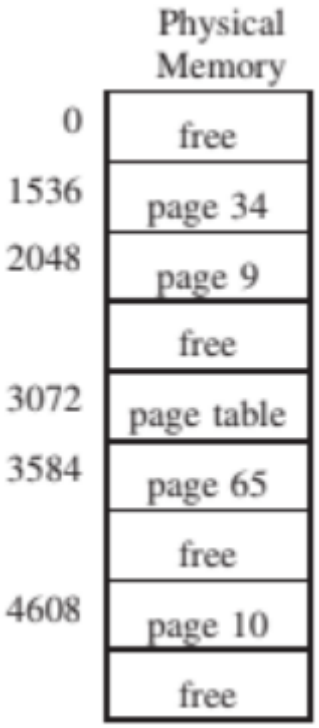
2.2 Problemas Conceptuales

Inciso 1

DATOS:
tamaño Página = 512 palabras
total Páginas (Memoria Virtual) = 512 páginas (0-511)
total Marcos (Memoria Física) = 16 marcos (0-15)
tamPagina=tamMarco

El tamaño de dirección física es 13 bits, entonces hay 2^13=8192 direcciones físicas.

Contenido Actual Memoria Física:



Inciso a)
Direcciones Físicas - Marco
(0 a 511) es el marco 0
(512 a 1023) es el marco 1
(1024 a 1535) es el marco 2
(1536 a 2048) es el marco 3
y así siguiendo…

En la imágen anterior, los números del lado izquierdo de la Memoria Física es la dirección Física en la que comienza ese marco. Entonces, para calcular en número de marco es:

$nroMarco = dirFis \,/\, tamMarco$ (división entera)

Calculamos los marcos en los que están las páginas cargadas en memoria física:
Página 34:
nroMarco = 1536/512 = 3

Página 9:
nroMarco = 2048 / 512 = 4

Página 65:
nroMarco = 3584 / 512 = 7

Página 10:
nroMarco = 4608 / 512 = 9
Luego, el contenido actual de la tabla de páginas es:

Tabla de Paginas	
nro Pagina	nro Marco
…	…
9	4
10	9
…	…
34	3
…	…
65	7
…	…

Inciso b)
Se carga la página 49 en la ubicación 0 y la página 34 es reemplazada por la página 12.

El contenido de la memoria física luego de los cambios es:

Memoria Fisica	
direccion Fisica	contenido Marco
0	page 49
...	free
1536	page 12
2048	page 9
...	free
3072	page table
3584	page 65
...	free
4068	page 10
...	free

El contenido de la tabla de páginas luego de los cambios es:

Tabla de Paginas	
nro Pagina	nro Marco
...	...
9	4
10	9
...	...
12	3
...	...
49	0
...	...
65	7
...	...

Inciso c)

Para traducir las direcciones virtuales a direcciones físicas, se debe realizar los siguientes pasos:

1-Obtener el número de página en el que está la dirección:
$nroPagina = dirVirtual\ div\ tamPag$ (división entera)

2-Obtener el offset dentro de esa página
$offset = dirVirtual\ mod\ tamPag$

3-Indexar la tabla de páginas por el nroPagina obtenido y conseguir el nroMarco correspondiente.

4-Luego, obtenemos la dirección física:
$dirFis = nroMarco * tamMarco + offset$

DIRECCIÓN VIRTUAL 4068
nroPagina = 4608 div 512 = 9
offset = 4608 mod 512 = 0
Según la tabla de páginas, la página 9 está en el marco 4
dirFis = 4*512+0 = 2048

DIRECCIÓN VIRTUAL 5119
nroPagina = 5119 div 512 = 9
offset = 5119 mod 512 = 511
Según la tabla de páginas, la página 9 está en el marco 4
dirFisica = 4*512+511=2559

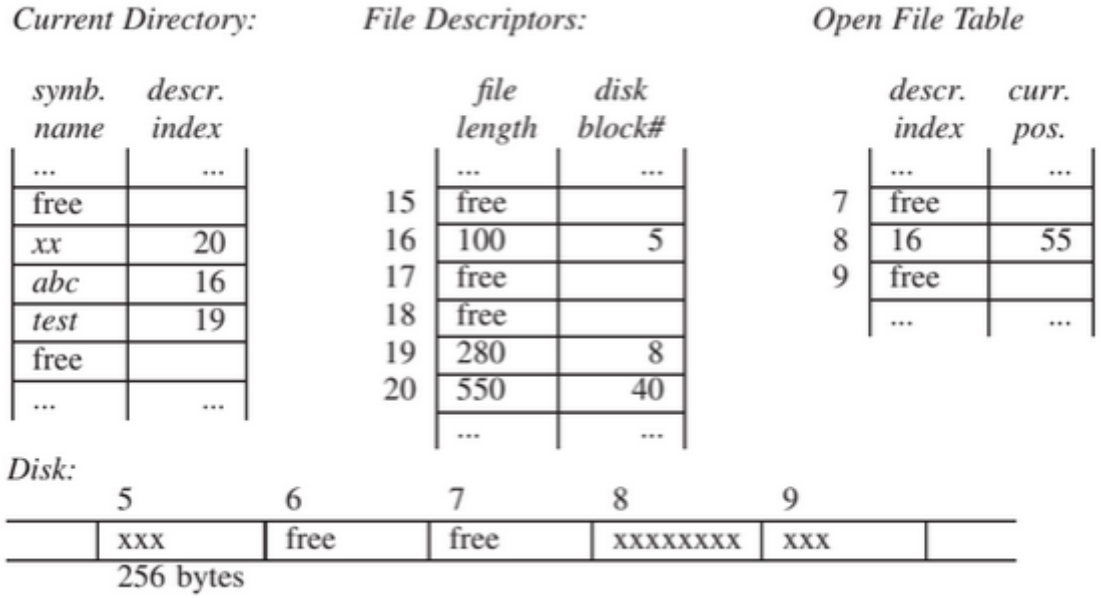
DIRECCIÓN VIRTUAL 5120
nroPagina = 5120 div 512 = 10
offset = 5120 mod 512 = 0
Según la tabla de páginas, la página 10 está en el marco 9
dirFis = 9*512+0 = 4608

DIRECCIÓN VIRTUAL 33300
nroPagina = 33300 div 512 = 65
offset = 33300 mod 512 = 20
Según la tabla de páginas, la página 65 está en el marco 7
dirFis = 7*512+20 = 3604

Inciso d)
La dirección virtual 33000 es referenciada. Entonces, traducimos a dirección física:

nroPagina = 33000 div 512 = 64
offset = 33000 mod 512 = 232
En la tabla de páginas, la entrada 64 no tiene ningún número de marco asignado. Entonces se produce un page fault. El sistema operativo toma control y el planificador de mediano término (swapper) trae la página 64 de memoria virtual a memoria física.

Inciso 2
Para el desarrollo del ejercicio se asume que los incisos a) a g) se ejecutan uno después de otro.
Contenidos iniciales de las estructuras de datos



Inciso a)

Abrir archivo test: fd=open(test)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
free		15	free		7	19	0
xx	20	16	100	5	8	16	55
abc	16	17	free		9	free	
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:							
	5	6	7	8	9		
	xxx	free	free	xxxxxxxx	xxx		

Se agrega en la tabla Open File Table una entrada con desc. index 19 y curr. pos en 0.

Inciso b)

Buscar la posición 60 del archivo abierto test: seek(fd,60)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
free		15	free		7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	free	
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:							
	5	6	7	8	9		
	xxx	free	free	xxxxxxxx	xxx		

Como el acceso es secuencial, en la entrada con desc. index=19 de Open File Table, el valor de curr. pos irá aumentando de a 1 hasta llegar a 60.

Inciso c)

Crear un archivo nuevo llamado new: create(new)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
new	15	15	0	6	7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	free	
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:

	5	6	7	8	9	
	xxx		free	xxxxxxxx	xxx	

En una entrada libre de Current Directory, se coloca symb name=new y se le asigna un file descriptor libre, en este caso 15.

En la entrada 15 de File Descriptors, se establece la longitud del archivo a 0 (porque al ser creado está vacío) y un número de bloque que esté disponible, en este caso 6.

El bloque 6 de disco está ocupado pero no tiene nada porque el archivo recién fue creado.

Inciso d)

Abrir el archivo new: fd=open(new)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
new	15	15	0	6	7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	15	0
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:

	5	6	7	8	9	
	xxx		free	xxxxxxxx	xxx	

En la tabla Open File Table, se agrega en una entrada disponible el desc. index en 15 y curr. pos en 0.

Inciso e)

Escribir 30 bytes en el archivo new: write(fd, buf,30)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
new	15	15	30	6	7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	15	30
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:

5	6	7	8	9	
xxx	x	free	xxxxxxxx	xxx	

En la tabla Open File Table, en la entrada en la que desc. index=15, queda curr. pos en 30. (Como el acceso es secuencial, este valor aumenta de a 1 desde 0 hasta llegar a 30)

También cambia la longitud de la entrada 15 en la tabla File Descriptors, de 0 a 30.

También se refleja el cambio en el bloque 6 de disco. Ahora no está vacío, ya que se escribieron 30 bytes.

Inciso f)

Cerrar el archivo new: close(fd)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
new	15	15	30	6	7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	free	
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:

5	6	7	8	9	
xxx	x	free	xxxxxxxx	xxx	

En Open File Table, se libera la entrada con desc. index en 15, ya que el archivo se cierra.

Inciso g)

Borrar el archivo new: delete(new)

Current Directory			File Descriptors			Open File Table	
symb name	desc. index		file length	disk block#		descr. index	curr. pos
...	...		...	...		...	...
free		15	free		7	19	60
xx	20	16	100	5	8	16	55
abc	16	17	free		9	free	
test	19	18	free			...	...
free		19	280	8			
...	...	20	550	40			
			...	...			

Disk:						
	5	6	7	8	9	
	xxx	free	free	xxxxxxxx	xxx	

Al borrar el archivo new, se libera la entrada de Current Directory que tenía symb name=new y desc. index=15

La entrada 15 en File Descriptors se libera, al igual que el bloque 6 del disco en el que estaba almacenado el archivo new.