



Proyecto Sistemas Operativos

Segundo Cuatrimestre 2024

Comisión 15 - Franco POPP - Matias David SCHWERDT

Índice

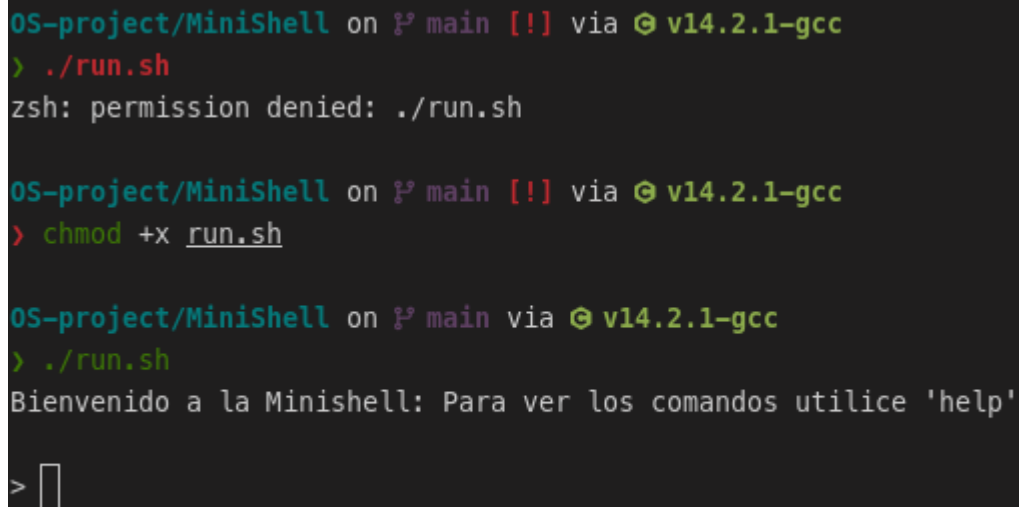
Aclaraciones.....	3
Pumper Nic.....	4
A) Implementación con procesos y pipes.....	4
B) Implementación con cola de mensajes.....	6
Mini Shell.....	8
Taller de motos.....	10
Santa Claus.....	12
Problemas Conceptuales.....	14
1).....	14
2).....	16

Aclaraciones

En todos los ejercicios de programación se ha incluido un script llamado “run.sh”.

Dicho script compila y ejecuta el ejercicio en cuestión.

En caso de no poder ejecutar dicho script por falta de permisos, deberá darle permiso de ejecución, tal como se ve en la siguiente imagen.



```
OS-project/MiniShell on ? main [!] via ? v14.2.1-gcc
> ./run.sh
zsh: permission denied: ./run.sh

OS-project/MiniShell on ? main [!] via ? v14.2.1-gcc
> chmod +x run.sh

OS-project/MiniShell on ? main via ? v14.2.1-gcc
> ./run.sh
Bienvenido a la Minishell: Para ver los comandos utilice 'help'

> 
```

Pumper Nic

A) Implementación con procesos y pipes

Pipes para Pedidos

1. **pipe_pedido_hamburguesa**: Comunica al cajero con el empleado que prepara las hamburguesas.
2. **pipe_pedido_vegano**: Similar al anterior, pero para pedidos de menús veganos.
3. **pipe_pedido_papas**: Utilizado para enviar pedidos de papas fritas a los 2 empleados que las preparan.

Pipes para Estado de Clientes

4. **pipe_hay_clientes**: Usado por los clientes para indicar al cajero que hay nuevos clientes esperando ser atendidos.

Pipes para Comunicación con Clientes

5. **pipe_cliente**: Utilizado por clientes regulares para enviar sus pedidos al cajero.
6. **pipe_clienteVIP**: Similar al anterior, pero exclusivo para clientes VIP.
Ambos pipes son no bloqueantes, para poder darle prioridad a los clientes VIP.

Pipes para Entrega de Pedidos

7. **pipe_hamburguesa**: Transporta las hamburguesas preparadas desde el empleado al cliente correspondiente.
8. **pipe_vegano**: Usado para la entrega de menús veganos completados.
9. **pipe_papas**: Para entregar pedidos de papas fritas terminados.

Procesos de Empleados

- **empleado1 (Hamburguesas)**: Se crea un proceso independiente utilizando fork(). Este proceso se encarga exclusivamente de preparar hamburguesas. Espera recibir un pedido a través de pipe_pedido_hamburguesa, prepara el pedido y lo envía al cliente correspondiente usando pipe_hamburguesa.
- **empleado2 (Menú Vegano)**: Similar al proceso de hamburguesas, este proceso maneja exclusivamente los pedidos de menús veganos. Utiliza pipe_pedido_vegano para recibir pedidos y pipe_vegano para enviar los pedidos completados.
- **empleado3 y empleado4 (Papas Fritas)**: Dos procesos separados se encargan de las papas fritas para manejar la alta demanda de este ítem. Ambos empleados reciben pedidos del mismo pipe_pedido_papas y entregan las papas fritas terminadas a través de pipe_papas.

Proceso del Cajero

- **cajero:** es un proceso que actúa como intermediario entre los clientes y los empleados. Este proceso recibe pedidos tanto de clientes regulares como VIP, priorizando éstos últimos. Los pedidos son recibidos a través de pipe_cliente y pipe_clienteVIP y redirigidos a los empleados correspondientes a través de los pipes de pedidos específicos. La implementación de pipes no bloqueantes para los clientes y clientes VIP asegura que los pedidos de estos últimos sean procesados con mayor prioridad.

Procesos de Clientes

- **Clientes Regulares y VIP:** Para cada cliente, ya sea regular o VIP, se crea un proceso. Estos procesos generan pedidos de manera aleatoria y los envían al cajero. Después de enviar sus pedidos, esperan recibir su pedido completado a través de uno de los pipes de entrega de los empleados.

Ejecución del Código

Para compilar y ejecutar el código, se incluye un archivo "run.sh".

```
schwerdt@raspberry:~/OS-project/Pumper-Nic/pipes $ ./run.sh
Cliente 5 llega al local
El cliente 5 pide Hamburguesa
Cliente 2 llega al local
El cliente 2 pide Menu vegano
Un cliente pidio: Hamburguesa
El cajero envia el pedido de hamburguesa al empleado 1
Un cliente pidio: Menu vegano
El cajero envia el pedido de menu vegano al empleado 2
El empleado 1 comienza a cocinar una hamburguesa
El empleado 2 comienza a cocinar un menu vegano
El empleado 1 entrega la hamburguesa
El empleado 2 entrega el menu vegano
El cliente 2 recibe Menu vegano
```

B) Implementación con cola de mensajes

Estructura de Procesos y Colas de Mensajes

Creamos un conjunto de procesos, cada uno encargado de diferentes tareas dentro del restaurante:

1. **Empleados (1, 2, 3 y 4):** Cada empleado es un proceso que se encarga de preparar un tipo específico de menú. Los empleados 1 y 2 preparan hamburguesas y menús veganos, respectivamente, mientras que los empleados 3 y 4 se ocupan de las papas fritas, siendo estas el ítem más solicitado.
2. **Cajero:** Este proceso actúa como intermediario entre los clientes y los empleados, gestionando los pedidos y enviándolos a los empleados correspondientes.
3. **Clientes (Regular y VIP):** Cada cliente es un proceso que envía pedidos al cajero y espera recibir su pedido completado. Los clientes VIP tienen prioridad sobre los clientes regulares.

Uso de Colas de Mensajes

El sistema utiliza una cola de mensajes identificada por una clave única (KEY). Esta cola es el medio principal de comunicación entre los clientes, el cajero y los empleados:

- **Envío de Pedidos:** Los clientes envían sus pedidos a la cola, donde cada mensaje contiene el tipo de menú solicitado, la identificación del cliente, y si el cliente es VIP o no. Los pedidos de clientes VIP se priorizan utilizando el tipo de mensaje en la cola.
- **Procesamiento de Pedidos:** El cajero recibe los pedidos desde la cola, revisando primero los mensajes de tipo VIP para asegurar servicio prioritario. Posteriormente, envía estos pedidos a los empleados correspondientes según el menú solicitado.
- **Preparación y Entrega:** Cada empleado recibe pedidos de su tipo específico de menú, los prepara, y una vez listos, los envía de vuelta a la cola de mensajes con el tipo de mensaje ajustado al ID del cliente, facilitando que el cliente reciba su pedido correcto.

Ventajas del Uso de Colas de Mensajes

Con una cola de mensajes, los procesos pueden enviar y recibir mensajes asincrónicamente, lo que significa que el emisor no tiene que esperar a que el receptor pueda procesar el mensaje.

Además, cuando los mensajes se envían a través de una cola, se almacenan en el sistema operativo, por lo que incluso si un proceso sale y se reinicia, aún se pueden recuperar los mensajes enviados. Esto aumenta la robustez y la tolerancia a fallos del sistema.

Un sistema con una cola de mensajes también puede asignar prioridades a los mensajes y ayudar a coordinar situaciones críticas, como pedidos VIP que merecen una respuesta inmediata.

Otra ventaja es que un mismo mensaje puede tener múltiples receptores. Con una cola de este tipo, varios empleados en diferentes ubicaciones pueden recibir el mismo mensaje.

El uso de colas de mensajes elimina la necesidad de manejar múltiples descriptores de archivo y la complejidad asociada con abrir, cerrar y mantener pipes entre procesos. En sistemas que utilizan pipes, es necesario cerrar adecuadamente los extremos no utilizados en cada proceso para evitar fugas de recursos y comportamientos inesperados, lo que puede complicar el código y hacerlo más propenso a errores.

Ejecución del Código

Para compilar y ejecutar el código, se incluye un archivo “run.sh”.

```
schwerdt@raspberrypi:~/OS-project/Pumper-Nic/cola-de-mensajes $ ./run.sh
El cliente VIP 500 pide Hamburguesa
Un cliente pidio: Hamburguesa
El cliente 501 pide Menu vegano
El cliente 502 pide Hamburguesa
El cliente 509 pide Menu vegano
El cliente 508 pide Papas Fritas
El cliente VIP 507 pide Menu vegano
El cliente 503 pide Menu vegano
El cliente 506 pide Menu vegano
El cliente VIP 505 pide Papas Fritas
El cliente VIP 504 pide Hamburguesa
El cajero envia el pedido de hamburguesa al empleado
Un cliente VIP pidio: Menu vegano
El empleado 1 comienza a cocinar una hamburguesa
El cajero envia el pedido de hamburguesa al empleado
Un cliente VIP pidio: Hamburguesa
El empleado 2 comienza a cocinar un menu vegano
El empleado 1 entrega la hamburguesa
El cliente VIP 500 recibe Hamburguesa
El cajero envia el pedido de hamburguesa al empleado
Un cliente VIP pidio: Menu vegano
```

Mini Shell

El archivo “main.c” implementa un bucle de lectura de comandos. Cada comando es ejecutado en un proceso hijo mediante fork. El proceso hijo invoca la función `execve`, que intenta ejecutar el comando buscándolo en la ruta “bin/comandos/”. El proceso padre espera a que el proceso hijo termine, antes de volver a iterar el bucle para solicitar un nuevo comando. El bucle termina cuando el usuario ingresa el comando “exit”, el cual finaliza la Mini Shell.

La Mini Shell incluye los siguientes comandos:

- **exit**: Sale de la Mini Shell.
- **mkdir**: Crea un directorio en la ruta especificada.
Guía de uso: `mkdir [ruta/Nombre_del_directorio]`
- **rm_dir**: Elimina un directorio en la ruta especificada.
Guía de uso: `rm_dir [ruta/Nombre_del_directorio]`
- **touch**: Crea un archivo en la ruta especificada.
Guía de uso: `touch [ruta/Nombre_del_archivo]`
- **ls**: Lista el contenido de un directorio.
Guía de uso: `ls [ruta/Nombre_del_directorio]`
- **cat**: Muestra el contenido de un archivo.
Guía de uso: `cat [ruta/Nombre_del_archivo]`
- **chmod**: Modifica los permisos de un archivo o directorio.
Los permisos pueden ser de lectura (+r, -r), escritura (+w, -w), y ejecución (+x, -x).
Guía de uso: `chmod [ruta/Nombre_del_archivo] [+r | -r | +w | -w | +x | -x]`

Ventajas de la implementación

La MiniShell está diseñada para ser extensible: basta con agregar nuevos ejecutables en la carpeta “bin/comandos/” para que puedan ser ejecutados directamente desde la MiniShell, sin necesidad de realizar cambios adicionales en el código fuente. Esta flexibilidad permite ampliar fácilmente las funcionalidades de la MiniShell a futuro.

Ejecución de MiniShell

Para simplificar la compilación y la organización de los ejecutables, creamos el script “run.sh”. Este script:

1. **Crea carpetas**: Verifica si las carpetas “bin” y “bin/comandos” existen, y las crea en caso de ser necesario.
2. **Compila archivos**:

```
• > tree
.
├── bin
│   └── comandos
│       ├── cat
│       ├── chmod
│       ├── help
│       ├── ls
│       ├── mkdir
│       ├── rm_dir
│       ├── rm_file
│       └── touch
└── main
    ├── cat.c
    ├── chmod.c
    ├── help.c
    ├── ls.c
    ├── main.c
    ├── mkdir.c
    ├── rm_dir.c
    ├── rm_file.c
    ├── run.sh
    └── touch.c
```


- Todos los archivos .c (excepto main.c) se compilan y se colocan en “bin/comandos”.
 - “main.c” se compila y su ejecutable se coloca en bin.
3. **Ejecución:** Finalmente, el script ejecuta el archivo main.

```
OS-project/MiniShell on P main [x?] via G v14.2.1-gcc
> ./run.sh
Bienvenido a la Minishell: Para ver los comandos utilice 'help'

> |
```

En la versión refactorizada del código, implementamos los comandos usando principalmente funciones de librerías.

Además, en caso de querer ejecutar un comando vacío, es decir, el usuario presiona la tecla Enter sin escribir ningún comando, ahora no ocurre nada. Simplemente se vuelve a solicitar al usuario que escriba un comando.

Ejemplos de ejecución de comandos

```
OS-project/MiniShell on P main via G v14.2.1-gcc
> ./run.sh
Bienvenido a la Minishell: Para ver los comandos utilice 'help'

> mkdir carpeta
El directorio fue creado exitosamente.

> chmod carpeta +r
Permisos cambiados correctamente.

> rm_dir carpeta
Directorio eliminado exitosamente.

> touch archivo
Archivo fue creado exitosamente.

> ls bin/comandos
.
touch
mkdir
rm_dir
cat
help
chmod
..
ls
>
```

```
> cat run.sh
#!/bin/bash

# Nombre del archivo a compilar y ejecutar
MAIN="main"

# Si no existe, se crea la carpeta "bin" para guardar el ejecutable
mkdir -p bin/comandos

# Compilar todos los archivos .c excepto el archivo main.c
for archivo in *.c; do
    if [ "$archivo" != "$MAIN.c" ]; then
        # Quitamos la extensión .c del nombre del archivo
        nombre_comando=$(basename "$archivo" .c)
        # Compilamos el archivo y lo colocamos en la carpeta bin/comandos/
        gcc "$archivo" -o "bin/comandos/$nombre_comando"
    fi
done

# Compilar el archivo main.c
gcc "$MAIN".c -o "bin/$MAIN"

# Ejecutar el archivo main
./bin/"$MAIN"
> exit
Gracias por usar la Minishell.
Autores: Franco Popp y Matias David Schwerdt.

OS-project/MiniShell on P main [?] via G v14.2.1-gcc took 1m28s
> |
```

Taller de motos

Cada operario utiliza semáforos para coordinarse, asegurando que todas las tareas se realicen en el orden adecuado. En la versión refactorizada del problema, cada operario sigue un "protocolo de dos fases", de forma que los métodos tengan el siguiente formato:

```
operario(){  
    // obtiene todos los semáforos necesarios  
    // ejecuta las acciones del operario  
    // libera todos los semáforos para los siguientes operarios  
}
```



Esto lo hicimos para no caer en el problema de tener que utilizar a un operario para sincronizar a los demás cuando este no tiene que realizar ninguna tarea.

Para esto, utilizamos 6 semáforos:

- **armar_ruedas**: inicia en 2, permitiendo armar ambas ruedas.
- **armar_cuadro** y **agregar_motor**: se inicializan en 0 y permiten que los operarios correspondientes solo avancen cuando termina el operario anterior.
- **pintar_moto**: se inicializa en 0 y sincroniza a los pintores para que sólo uno pinte la moto de verde o rojo.
- **equipar_moto**: se inicializa en 1 para que desde le primer ciclo haya mejora, aunque podría comenzar en 0, permite que una de cada dos motos obtenga una mejora.
- **sem_ciclo**: permite sincronizar los ciclos de mejora/no-mejora, su función es no permitir que el operario 1, encargado de armar las ruedas, comience a ejecutar luego de que el pintor habilite el semáforo, pero antes de que se realice la mejora en la iteración que corresponde. El operario de mejoras hará 4 signals para este semáforo, de forma que se habilite a las 4 ruedas que se deben armar hasta la próxima vez que se ejecute el método de equipar mejora.

Ejecución del código de taller de motos

Para compilar y ejecutar el código, se incluye un archivo “run.sh”.

```
OS-project/Taller-de-motos on  main via  v14.2.1-gcc
❌ > ./run.sh
Operario 1: poniendo una rueda...
Operario 1: poniendo una rueda...
Operario 2: armando cuadro...
Operario 3: agregando motor...
Operario 4: pintando moto de verde...
Operario 6: equipando mejora...
Operario 1: poniendo una rueda...
Operario 1: poniendo una rueda...
Operario 2: armando cuadro...
Operario 3: agregando motor...
Operario 5: pintando moto de rojo...
Operario 1: poniendo una rueda...
Operario 1: poniendo una rueda...
Operario 2: armando cuadro...
Operario 3: agregando motor...
Operario 4: pintando moto de verde...
Operario 6: equipando mejora...
Operario 1: poniendo una rueda...
Operario 1: poniendo una rueda...
Operario 2: armando cuadro...
Operario 3: agregando motor...
Operario 5: pintando moto de rojo...
Operario 1: poniendo una rueda...
Operario 1: poniendo una rueda...
```

Santa Claus

Semáforos

- **sem_despierta_santa:** Inicializado en 0. Este semáforo se usa para despertar a Santa Claus. Se incrementa cuando un grupo de tres elfos necesita ayuda o cuando el noveno reno llega.
- **Semáforos de elfos:**
 - **sem_elfo_vacio:** Inicializado en 2. Controla cuántos elfos con problemas pueden acercarse a la tienda de Santa. Los primeros dos elfos logran reducir el contador, y el tercero no puede esperar y despierta a Santa.
 - **sem_elfo_lleno:** Controla cuántos elfos han sido atendidos. Cada elfo en el grupo de tres espera en este semáforo hasta recibir ayuda.
 - **sem_elfo_grupo:** Inicializado en 3. Permite que solo un grupo de tres elfos se acerque a Santa a la vez. Los otros elfos que encuentren problemas deben esperar hasta que el grupo actual reciba ayuda.
 - **sem_elfo_ayudado:** Cada elfo en el grupo de tres espera aquí para recibir la ayuda de Santa.
- **Semáforos de renos:**
 - **sem_reno_vacio:** Inicializado en 8. Controla el grupo de renos; el noveno reno que llega despierta a Santa.
 - **sem_reno_grupo:** Inicializado en 9. Asegura que todos los renos se agrupen antes de que uno despierte a Santa.
 - **sem_reno_ayudado:** Cada reno espera aquí hasta recibir la ayuda de Santa para ser enganchado al trineo.

Mutex

- **mutex_elfo:** Controla el acceso a las variables de los elfos. Esto evita que más de tres elfos intenten acercarse a Santa al mismo tiempo y asegura que solo un grupo de tres pueda solicitar ayuda. Evitamos condición de carrera.
- **mutex_reno:** Controla el acceso a las variables de los renos. Esto asegura que solo el último reno en llegar despierte a Santa y que los otros renos esperen sin interferir. Evitamos condición de carrera.

Threads

- **Hilo de Santa Claus (t_santa):** Este hilo representa a Santa y maneja las acciones cuando es despertado por los renos o elfos. Solo se despierta cuando:
 - Todos los renos están presentes (9 en total).
 - Tres elfos tienen problemas y lo necesitan. Santa prioriza ayudar a los renos antes que a los elfos.
- **Hilos de renos (t_reno[]):** en total son 9 hilos de renos. Cuando todos los renos llegan, el último reno despierta a Santa Claus.

- **Hilos de elfos (t_elfo[]):** en total son 6 hilos de elfos. Cada hilo de elfo simula el trabajo de un elfo. Si un elfo encuentra un problema, espera hasta que haya un grupo de tres elfos con problemas para despertar a Santa y recibir ayuda.

Explicación del Funcionamiento

1. **Renos:** Cada reno se va sumando al grupo. Cuando el noveno reno llega, adquiere mutex_reno, intenta decrementar sem_reno_vacio (que fallará ya que todos los renos están presentes) y despierta a Santa. Luego, espera en sem_reno_ayudado para ser enganchado al trineo.
2. **Elfos:** Cada elfo que encuentra un problema intenta formar un grupo de tres. Los primeros dos elfos disminuyen sem_elfo_vacio, y el tercero despierta a Santa y esperan en sem_elfo_lleno. Luego de ser ayudados, los elfos en este grupo incrementan sem_elfo_grupo para permitir que otro grupo de tres elfos pueda pedir ayuda.
3. **Santa:** Santa despierta cuando sem_despierta_santa se activa. Primero revisa si todos los renos están presentes, en cuyo caso los prioriza y los engancha al trineo. Si no están todos los renos, atiende al grupo de elfos en espera.

Ejecución del Código

Para compilar y ejecutar el código, se incluye un archivo "run.sh".

```
OS-project/Santa-Claus on ? main [!] via G v14.2.1-gcc
> ./run.sh
Elfo tiene un problema
Elfo trabaja sin problemas
Elfo trabaja sin problemas
Elfo tiene un problema
Elfo trabaja sin problemas
Elfo trabaja sin problemas
Elfo trabaja sin problemas
Elfo trabaja sin problemas
Elfo tiene un problema
Elfo trabaja sin problemas
Elfos: Somos 3 elfos, queremos despertar a Santa
Santa se despierta
Santa ayuda a los Elfos
Reno llega. Espera en la cabaña
Reno llega. Espera en la cabaña
Reno llega. Espera en la cabaña
Reno llega. Espera en la cabaña
```

Problemas Conceptuales

1)

a)

Datos:

Dirección lógica de 16 bits: 0011 0000 0011 0011

Tamaño de página: 512 direcciones

Número de página = P

Número de marco = M = P/2

Una dirección lógica está compuesta por un número de página y un desplazamiento.

Para saber el tamaño de página, hacemos el siguiente cálculo $\log_2 512 = 9$.

Como mi dirección lógica es de 16 bits, de los cuales 9 son utilizados para el desplazamiento, me quedan 7 bits para el número de página.



Por lo tanto de la dirección lógica 0011 0000 0011 0011 tenemos que:

- Número de página = $0011000_2 = 24_{10}$
- Desplazamiento = $000110011_2 = 51_{10}$

La **dirección física** está formada por un número de marco y el desplazamiento.

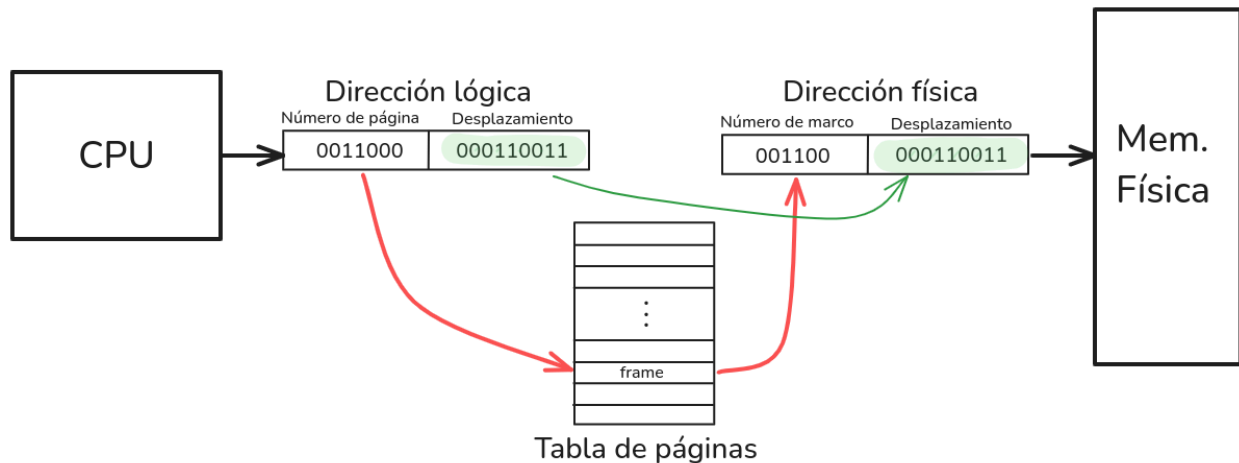
Por el enunciado sabemos que el número de marco es la mitad del número de página:

Número de marco = Número de página / 2

Número de marco = $24 / 2$

Número de marco = 12

El desplazamiento es el mismo tanto en la dirección lógica como en la física.



Este diagrama lo hemos realizado con la siguiente página web <https://excalidraw.com/>

b)

Datos:

Dirección lógica de 16 bits: 0011 0000 0011 0011

Tamaño máximo de segmento: 2K direcciones = 2048 direcciones

Direcciones reales: 20 + 4096 + Número de Segmento

Como $\log_2 2048 = 11$, entonces 11 bits representan el desplazamiento.

La dirección lógica es de 16 bits, así que los 5 bits restantes representan el número de segmento.

Número de segmento	Desplazamiento
5 bits	11 bits

Por lo tanto de la dirección lógica 0011 0000 0011 0011 tenemos que:

- Número de segmento = $0011_2 = 6_{10}$
- Desplazamiento = $00000110011_2 = 51_{10}$

La dirección base del segmento está determinada por la fórmula:

$$\text{Base} = 20 + 4096 + \text{Número de segmento}$$

$$\text{Base} = 20 + 4096 + 6$$

$$\text{Base} = 4122$$

La dirección física se obtiene sumando la dirección base del segmento y el desplazamiento dentro del segmento.

- Dirección base (en decimal): 4122
- Desplazamiento: 51

$$\text{Dirección física} = 4122 + 51 = 4173$$

Convertimos la dirección física decimal 4173 a binario:

$$4173_{10} = 0001000001001101_2$$

2)

a)

De dirección virtual a dirección física

Para convertir una dirección virtual en una dirección física en sistemas que tienen paginación, inicialmente se descompone la dirección virtual en dos partes: el número de página y el desplazamiento. El número de página se determina a partir de los bits más significativos de la dirección, mientras que el desplazamiento se obtiene de los bits menos significativos.

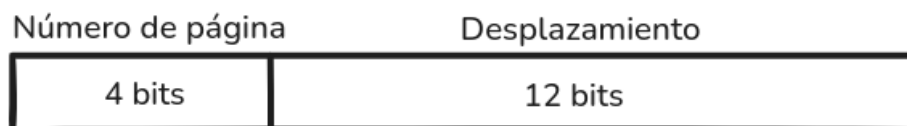
Una vez identificado el número de página, se consulta la tabla de páginas para encontrar el marco de página correspondiente en la memoria física. La tabla de páginas es esencial para mapear los números de páginas virtuales a los marcos físicos.

Finalmente, la dirección física se forma con el número de marco y el desplazamiento.

Datos

Cada página tiene un tamaño de 4096 bytes. Para saber cuantos bits le corresponde al desplazamiento dentro de la página, hacemos $\log_2 4096 = 12$.

Sabemos que la dirección virtual es de 16 bits, de los cuales 12 bits son para el desplazamiento, por lo tanto nos quedan 4 bits para el número de página.



- $0x621C_{16} = 0110\ 0010\ 0001\ 1100_2$
 Número de página = $0110_2 = 6_{16} = 6_{10}$

Desplazamiento = $0010\ 0001\ 1100_2 = 21C_{16} = 540_{10}$

Según la tabla de páginas, el número de marco asociado al número de página 6, es 8.
Por lo tanto la dirección física es **0x821C**. El bit de referencia se actualiza con un 1.

- $0xF0A3_{16} = 1111\ 0000\ 1010\ 0011_2$
Número de página = $1111_2 = F_{16} = 15_{10}$
Desplazamiento = $0000\ 1010\ 0011_2 = 0A3_{16} = 163_{10}$

Según la tabla de páginas, el número de marco asociado al número de página 15, es 2.
Por lo tanto la dirección física es **0x20A3**. El bit de referencia se actualiza con un 1.

- $0xBC1A_{16} = 1011\ 1100\ 0001\ 1010_2$
Número de página = $0110_2 = B_{16} = 11_{10}$
Desplazamiento = $1100\ 0001\ 1010_2 = C1A_{16} = 3098_{10}$

Según la tabla de páginas, el número de marco asociado al número de página 11, es 4.
Por lo tanto la dirección física es **0x4C1A**. El bit de referencia se actualiza con un 1.

- $0x5BAA_{16} = 0101\ 1011\ 1010\ 1010_2$
Número de página = $0101_2 = 5_{16} = 5_{10}$
Desplazamiento = $1011\ 1010\ 1010_2 = BAA_{16} = 2986_{10}$

Según la tabla de páginas, el número de marco asociado al número de página 5, es 13.
Por lo tanto la dirección física es **0xDBAA**. El bit de referencia se actualiza con un 1.

- $0x0BA1_{16} = 0000\ 1011\ 1010\ 0001_2$
Número de página = $0000_2 = 0_{16} = 0_{10}$
Desplazamiento = $1011\ 1010\ 0001_2 = BA1_{16} = 2077_{10}$

Según la tabla de páginas, el número de marco asociado al número de página 0, es 9.
Por lo tanto la dirección física es **0x9BA1**. El bit de referencia se actualiza con un 1.

La tabla de páginas finalmente queda así:

Página	Marco	Bit de referencia
0	9	1
1	-	0
2	10	0
3	15	0

4	6	0
5	13	1
6	8	1
7	12	0
8	7	0
9	-	0
10	5	0
11	4	1
12	1	0
13	0	0
14	-	0
15	2	1

b)

Un fallo de página ocurre cuando una dirección virtual referencia una página que no está actualmente cargada en memoria física. Una página no está en memoria si el marco asociado muestra un guión ("-").

Un ejemplo de fallo de página podría ser **0x1010** donde:

- El número de página es 1.
- El desplazamiento es 010.

c)

En el caso de un fallo de página, el algoritmo LRU (Least Recently Used) elegirá la página del conjunto de páginas que no han sido recientemente usadas. Para determinar esto, se utiliza el bit de referencia de la tabla de páginas.

En este ejercicio, las páginas 1, 9 y 14 no necesitan reemplazo. Y de las restantes páginas, las que tienen bit de referencia en 0 son las páginas 2, 3, 4, 7, 8, 10, 12 y 13. Por lo tanto cualquiera de ellas podría ser reemplazada.