Informe Proyecto

Comisión 16 2024 EDITION

Thiago Trotta • Lucio Rodríguez

Si compila está bien, si arranca es perfecto.

Linus Torvalds

Contenido

Parte Uno ⬛ Experimentación de Procesos y Threads con los SO

1. Procesos, threads y comunicación

2. Sincronización

Parte Dos ⬛ Problemas

1. Lectura

2. Problemas Conceptuales

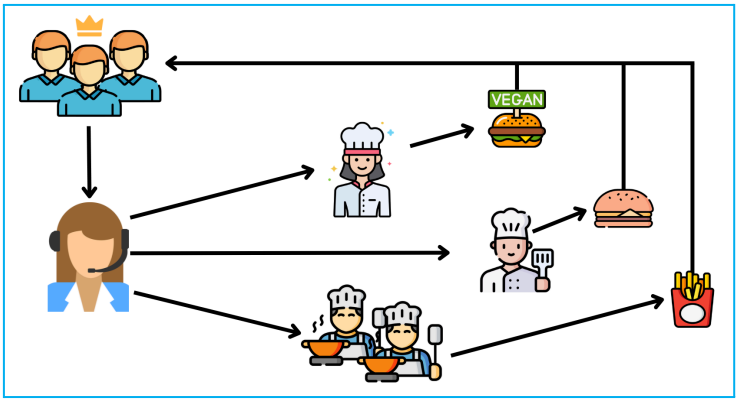
Ejecución de los programas

1. Experimentación de Procesos y Threads con los Sistemas Operativos

1.1. procesos-threads-y-comunicación

1.1.1. Pumper Nic.

Para el problema de la cadena de comida rápida Pumper Nic., tomamos la decisión que el flujo por el cuál se iba a dar la secuencia de atención, preparación y recibo de los pedidos se respete de la misma forma tanto para su implementación mediante pipes, como colas de mensajes. En ambos casos, la situación de cola llena se simuló usando un número aleatorio que determina si el cliente intenta nuevamente más tarde. De esta manera, es mucho mejor notar las diferencias entre cada una y las ventajas y desventajas que conlleva implementar la solución con las distintas comunicaciones entre los procesos. El flujo básico de la atención de un pedido es el siguiente:



Los clientes llegan (pueden ser VIP o normales) y eligen uno de los tres combos posibles. El despachador recibe la orden, identifica el tipo de combo (hamburguesa, menú vegano o papas fritas) y notifica al cocinero correspondiente para que lo prepare. Una vez listo, el pedido se comunica al cliente y tomamos la decisión de que se recoja en un área de entrega, por lo que una vez terminado el cocinero, el pedido va directamente al cliente.

a. Pipes

Para la implementación con pipes se hizo uso de 8 pipes, los cuáles cada uno representa una flecha del gráfico de arriba, a excepción del cliente-despachador donde contamos con dos, uno para los clientes normales y otro para los vip. Cada pipe se nombra de acuerdo con las entidades involucradas (por ejemplo, pipe_desp_papas para el despachador y el empleado de papas fritas).

Para manejar la prioridad de los clientes VIP sobre los normales, se crearon dos pipes: pipe_vip_desp y pipe_normal_desp. Al crearse un cliente, se determina si es VIP o no, y según eso se comunica con su canal correspondiente. El despachador primero intenta leer de pipe_vip_desp y luego de pipe_normal_desp.

Sin embargo, hacer ambos pipes no bloqueantes generaría espera ocupada, lo cual queremos evitar. Por eso, solo el pipe de los VIP es no bloqueante. Así, si hay clientes VIP,

estos son atendidos primero, y cuando no hay, el despachador pasa al pipe de clientes normales, que es bloqueante.

La limitación que tiene esta estrategia es que, si el despachador está bloqueado en el pipe de clientes normales y llegan nuevos VIP, estos deberán esperar hasta que llegue un cliente normal para que el despachador pueda volver a revisar el pipe de VIP. El peor caso a su vez que puede pasar es que si te llegan todos VIP o los últimos son VIPs, no va a haber progreso para estos clientes y van a quedar bloqueados sin nunca poder avanzar. Otra de las limitaciones es que con la entrega del pedido, como todos los clientes comparten el mismo pipe para ese pedido, no te aseguras nunca que el primero que pidió sea el primero en recibir, por lo que puede llegar a pasar que uno pida una hamburguesa, y en ese instante ve que en el pipe_hamb_cli trajo una hamburguesa y la agarra. Esto haría que el primero que pidió siga esperando.

b. Cola de mensajes

Implementar la solución con una cola de mensajes resultó más manejable y se adaptó mejor a los requerimientos. Pudimos modularizar el sistema, organizando cada proceso en archivos ejecutables independientes. Las constantes de tipo de mensaje y el uso de un struct facilitaron la identificación de los pedidos. Al asignar un tipo de mensaje a cada combo y cliente, el despachador puede enviar mensajes específicos al empleado adecuado (por ejemplo, "papas fritas" tiene el tipo 5) y recibir mensajes de los clientes con prioridades: **CLIENTE_VIP** tiene el tipo 1 y **CLIENTE_NORMAL**, el 2.

La función msgrcv con un msg_type negativo nos permitió priorizar mensajes de clientes VIP, que son atendidos primero si están en cola. entonces se accede al primer mensaje cuyo tipo sea menor o igual al valor absoluto y a la vez sea el menor de todos. Entonces el valor que recibe la cola en despachador es -2, por lo que al calcular el valor absoluto que es 2, le estamos indicando que sólo reciba mensajes de tipo 1 y 2, y tenga prioridad primero que todo que con 1 (vips).

Además, en esta implementación el cocinero puede enviar el pedido terminado directamente al cliente correcto que lo pidió mediante la id_recibir, a diferencia de los pipes, donde el pedido lo tomaba el cliente disponible más rápido. En el código, sumamos un diferenciador a id_recibir para evitar conflictos con los mensajes de la lógica principal (1 a 5). Esto asegura que, por ejemplo, un cliente con id 3 no se confunda con el mensaje destinado al preparador de hamburguesas.

Conclusión: Implementar la solución con colas de mensajes resultó ser mucho más ventajoso que con pipes, debido a su flexibilidad. Las colas permitieron manejar prioridades de manera natural, respetando la regla de atender primero a clientes VIP sin problemas de bloqueos indefinidos, como ocurría con los pipes. Además, cada pedido pudo asignarse correctamente al cliente correspondiente, en lugar de que cualquier cliente lo tome, como pasaba en pipes. Finalmente, el uso de una sola cola de mensajes en común a la hora de poner en práctica la solución fue mucho mejor que tener que crear una gran cantidad de pipes y cerrarlos cada vez que no los usábamos.

1.1.2. Mini shell

La mini shell implementada nos permite ejecutar una serie de comandos básicos para gestionar archivos y directorios. Los comandos requeridos por el enunciado los implementamos con los siguiente acrónimos:

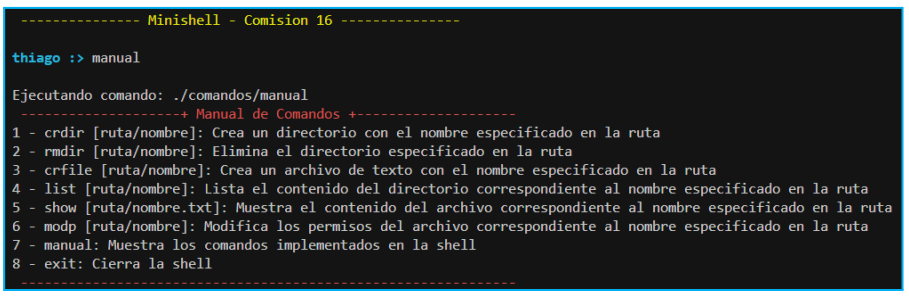
- **manual**: Muestra una lista de los comandos disponibles y sus descripciones.
- **crdir**: Crea uno o más directorios especificados por el usuario.
- **rmdir**: Elimina uno o más directorios especificados.
- **crfile**: Crea archivos de texto con los nombres y rutas especificadas.
- **list**: Muestra el contenido de los directorios especificados.
- **show**: Muestra el contenido de los archivos de texto especificados.
- **modp**: Modifica los permisos de los archivos especificados, permitiendo configuraciones como lectura, escritura y ejecución.
- **exit**: Finaliza la ejecución del shell.

Al nivel de código, la ejecución de los comandos se gestiona a través de la función ejecutarComando(), que utiliza fork() para crear un proceso hijo en el que se ejecutan los comandos. Esto permite que el shell continúe operando mientras los comandos se ejecutan de forma independiente. La función execv() se utiliza para ejecutar los comandos desde rutas específicas (./comandos/), garantizando que solo se ejecuten comandos diseñados para este shell.

El programa también incluye mecanismos para validar la entrada del usuario, asegurando que los comandos no excedan el límite de caracteres permitido (MAX_CHAR). Además, el salto de línea (\n) al final de la entrada se elimina para evitar problemas durante el análisis de los argumentos. Los comandos y sus argumentos se procesan usando strtok() y se almacenan en el arreglo args para su posterior ejecución.

Uno de los aspectos más destacados de esta implementación es su enfoque modular, que facilita la adición de nuevos comandos. Para agregar un comando adicional, basta con implementarlo en la carpeta de comandos, y la función ejecutarComando() lo detectará y ejecutará automáticamente. Sin embargo, una limitación de esta implementación es que las ayudas del manual deben ser hardcodeadas, lo que impide una automatización total en la incorporación de nuevos comandos al manual. A pesar de esto, la estructura actual permite una expansión sencilla y flexible de la funcionalidad.

Para los primeros usuarios de esta mini shell, se recomienda utilizar el comando manual, ya que proporciona una guía completa y un pantallazo inicial sobre los comandos disponibles y sus funcionalidades.



1.2. sincronización

1.2.1. Taller de motos

Para la implementación del problema, utilizamos 6 hilos, uno para cada operario. En cuanto a los semáforos, empleamos un total de 6: cada operario tiene un semáforo asignado para coordinar el inicio de su tarea, a excepción de los pintores, que comparten el mismo semáforo para permitir que cualquiera de los dos tome la moto y la pinte en su turno. Luego agregamos otro semáforo de control para que la secuencia se pueda manejar de manera correcta. Dado que las tareas deben realizarse de una en una y cada operario debe esperar su turno antes de comenzar, el problema tiene una naturaleza secuencial. Por lo tanto, la secuencia de trabajo puede pensarse como AABC(DvE)AABC(DvE)F…, donde:

- A representa al operario que arma las ruedas.
- B representa al operario que arma el cuadro.
- C representa al operario que instala el motor.
- (DvE) representa a los dos pintores, donde solo uno de ellos pinta la moto (verde o rojo).
- F representa al operario que agrega el equipamiento extra.

En nuestra solución, se estableció que la primera moto que esté lista no recibirá el equipamiento extra, mientras que la siguiente sí, y así sucesivamente. Este comportamiento puede invertirse simplemente modificando la inicialización de los semáforos, tal como se comenta en el código. Esto se logra mediante la sincronización adecuada con los semáforos, garantizando que las motos se fabriquen en orden y se respete la alternancia en el equipamiento.

Para lograrlo, utilizamos un semáforo de control llamado semControl. Este semáforo regula el flujo de fabricación desde el inicio del hilo encargado de las ruedas, donde se realiza un wait(semControl) antes de continuar. Al inicializar semControl en 4 (en una vuelta se consumen 2 por las 2 ruedas), permitimos que, en las dos primeras iteraciones, los operarios puedan avanzar directamente a la fabricación de ruedas sin interrupciones. Los indicados de dar paso a que se agreguen las ruedas son los pintores con los dos post de semRueda y desde el equipamiento extra se decide cuando dejar paso a que se consuman esos valores desde el wait(semRueda).

Cuando llega el momento de agregar el equipamiento extra (en la segunda iteración), aunque los pintores envíen dos señales (post a semRueda como dijimos), el avance queda bloqueado en semControl. Esto asegura que el operario encargado del equipamiento extra tenga tiempo para trabajar sin interferencias. Una vez que el equipamiento extra se ha agregado, este operario restablece el flujo liberando cuatro señales post(semControl) cuatro veces, permitiendo que se inicie la fabricación de la siguiente moto.

1.2.2. Santa Claus

El modelo implementado para resolver el problema utiliza un hilo para representar a Santa, y otros nueve y diez hilos representando a los renos y elfos respectivamente. Luego se definen varios semáforos la sincronización entre los hilos y mutex para proteger las secciones críticas. El semáforo *semSanta* controla cuándo Santa debe despertarse, es decir que tanto elfos como los renos pueden enviar esta señal en cuánto necesiten de él. Sin embargo, si ambos lo van a despertar al mismo tiempo, Santa tiene prioridad por sobre los renos ya que pregunta primero por ellos. Esto se logra por los semáforos contadores *cantElfos* y *cantRenos* que controlan en todo momento la cantidad de elfos/renos listos para despertar a Santa, ya que al estar inicializados en 9 y 3, si está en 0 algún semáforo es porque ya todos están listos. Es por eso que desde el hilo de Santa sólo se utiliza la cantidad de renos para preguntar, ya que si estos están en 0 los atiende primero, pero si no lo están los renos, entonces necesariamente tienen que ser los elfos quien los despertaron producto de llamarlos con el mismo semáforo.

Respecto a esto último, tomamos esta decisión en el diseño, ya que otra forma era realizar en vez del else donde trabajamos con los elfos, hacer otro if preguntando por los *cantElfos*. El problema que tenía esto, es que si despertaban a Santa los dos a la vez entonces con una sola vuelta a santa ya se ayudaban a ambos. Entonces de esta forma tomamos la convención de que santa sólo los puede ayudar de a uno por cada vuelta que hace en el ciclo.

Luego *semRenos* y *semElfos* gestionan la disponibilidad de los renos y los elfos, donde al final de renos() se aprecia que los animales están esperando con un wait(*semRenos*) a que santa les indique que vuelvan una vez hecho su trabajo a tener vacaciones (en la solución asumimos que al principio todos los renos están de vacaciones). La lógica para los elfos sin embargo es levemente distinta, ya que la cantidad máxima para ir a despertar a Santa no es la misma que la cantidad de hilos en total. Es decir tenemos 10 elfos y con 3 en problemas nos basta. Para ello se usa *semElfos* inicializado en 1 que simula que si es activado es porque un elfo tiene problemas. Es decir todos los elfos que llegan se quedan al principio de elfos() en un wait(*semElfos*), cuando uno los tiene decrementa la cantidad de elfos y si es el último despierta a Santa. Luego existe un semáforo extra para los tres elfos que fueron con Santa que están al final del código como los renos, haciendo un wait(*semElfosSanta)* que una vez los elfos pudieron resolver todo con Santa, se hacen tres post de este semáforo desde santa justamente.

El hecho de tener prioridad en los renos, y hacerlo de la manera en que lo implementamos se puede apreciar en algunos casos inanición para los elfos. Esto se debe a que por más que los elfos puedan querer despertar a Santa, los renos van a entrar primero antes de que lo puedan hacer los elfos. Para solucionar este tema requeriría complejizar aún más el algoritmo y seguramente agregar algún semáforo extra. Una de las opciones por las que podemos optar es la de simular la llegada de vacaciones de los renos cada un determinado tiempo utilizando un sleep. Sabemos que este problema NO se soluciona de esta manera, sin embargo se puede usar para apreciar mejor cómo ambas entidades actúan con Santa. Aún así cuando se ejecuta sin ningún sleep se puede ver como la solución planteada actúa correctamente, ya que de a momentos si pueden ser atendidos los elfos.

2. Problemas

2.1. Lectura

Para la actividad de lectura nos tocó investigar sobre la temática de Seguridad respecto a los fallos de confidencialidad. La propuesta que decidimos hacer fue mediante unas diapositivas las cuales se pueden apreciar algunas imágenes previas, y utilizamos algunos ejemplos de la confidencialidad en CrowdStrike.

Bibliografía utilizada:

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts* (Wiley, 10th Edition), Chapter 16: Security
- CrowdStrike - Identity Segmentation. Fuente: https://www.crowdstrike.com/en-us/cybersecurity-101/identity-protection/identity-segmentation/

2.2. Problemas conceptuales

2.2.1. Sistema de paginación y segmentación

Datos generales del problema:

- Dirección lógica de 16 bits
- Buscamos traducir la siguiente dirección: 0011000000110011_2

a. Sistema de paginación

Datos:

- Tamaño de página: 512 direcciones
- Número de marco: M = P/2

El tamaño de página es de 512 direcciones, lo que equivale a 2^9 direcciones por página. Por lo tanto, se necesitarán 9 bits de offset de la dirección lógica representan el desplazamiento dentro de la página y los bits restantes entonces representarán el número de página. Cómo la dirección lógica cuenta con 16 bits entonces resulta sencillo sacar el valor del número de páginas, 16 - 9 = 7 bits para el número de página. Esto es 2^7 = 128 páginas.

Ahora podemos trabajar con la dirección binaria para poder traducirla. La misma entonces estaría dividida de la siguiente manera:

0011000	000110011
Nro. Página	Offset

Luego los valores en decimal son: Página = 24_{10}, Desplazamiento = 51_{10}.

Con estos valores podemos armar la dirección física. El tamaño de la página y del marco es el mismo, por lo que sólo queda averiguar el número de marco. Este por la fórmula dada es la mitad del número de página (M = P / 2), entonces:

Número de marco M = 24/2 = 12

Ahora tomamos el número de marco y lo combinamos con el desplazamiento para formar la dirección física.

Dirección física $_{10}$: $12 \times 512 + 51 = 6144 + 51 = 6195_{10}$
Dirección física $_2$: $0001100\ 000110011_2$

b. Sistema de segmentación

Datos:

- Tamaño máximo de segmento: 2K direcciones = 2 * 1024 = 2048 direcciones
- Direcciones reales: 20 + 4,096 + Nro Segmento

Como cada segmento puede tener hasta 2048 = 2^{11} direcciones, entonces nos alcanza para saber que el desplazamiento dentro del segmento tendrá 11 bits. Luego los otros primeros bits de la dirección lógica los sacamos sabiendo que la dirección tiene 16 bits. 16 - 11 = 5 bits de segmento.

Ahora podemos trabajar nuevamente con la dirección binaria para poder traducirla. La misma entonces estaría dividida de la siguiente manera:

00110	000000110011
Nro. Segmento	Offset

Luego los valores en decimal son: Segmento = 6_{10}, Desplazamiento = 51_{10}.

Según el problema, las bases se colocan en las direcciones reales que se calculan con la fórmula dada en los datos. Luego para el segmento 6:

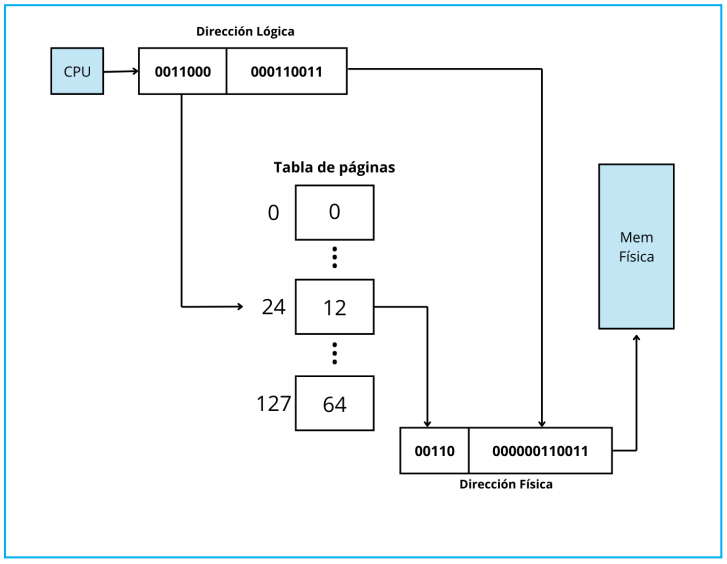
$$\text{Dirección real} = 20 + 4096 + 6 = 4122_{10}$$

Una vez calculado estos datos, sumamos entonces la base del segmento y el desplazamiento el cual nos debería dar la traducción final a la dirección física al convertirla en binario:

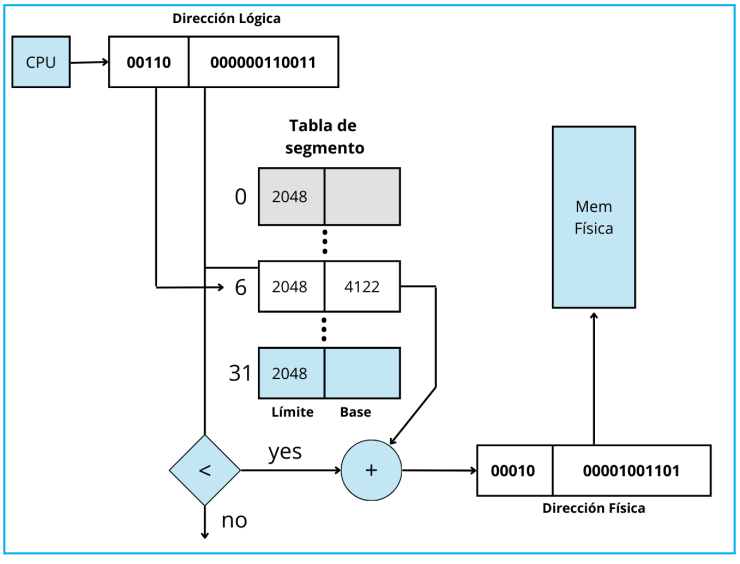
Dirección física $_{10}$: $4122 + 51 = 4173_{10}$

Dirección física $_{2}$: $00010\ 00001001101_{2}$

a



b



2.2.2. Traducción mediante tabla de páginas

Datos generales del problema:

- Sistema con direcciones virtuales y físicas de 16 bits
- Páginas de 4096 bytes

Dado que las páginas tienen un tamaño de 4,096 = 2^{12} bytes, entonces podemos obtener la cantidad de bits necesarios para el desplazamiento de la dirección lógica (también para la dirección física ya que es el mismo). Entonces, los últimos 12 bits serán el offset dentro de la página, y los 4 bits restantes (16 - 12 = 4) representarán el número de página.

En base a los datos obtenidos, para convertir una dirección virtual a una dirección física en este sistema requerimos de los siguiente pasos:

1. Obtenemos el número de página a partir de los 4 bits más significativos.
2. Obtenemos el desplazamiento a partir de los 12 bits menos significativos.
3. Consultamos la tabla de páginas para localizar la página correspondiente al número de página (requiere traducirla a decimal).
4. Si la página en la tabla de páginas tiene asociado un número de marco, entonces se encuentra en memoria. Luego utilizamos el número de marco correspondiente a la página y el desplazamiento para realizar la traducción.
5. Si no está en memoria porque en la tabla de páginas hay un -, tenemos un fallo de página (*page fault*).

Traducción: La dirección física si trabajamos en binario es la concatenación del número de marco más el desplazamiento. Cómo la dirección está dividida en 4 y 12, entonces trabajando con hexadecimales podemos usar la misma estrategia ya que justamente un número en hexadecimal se puede representar con cuatro dígitos en binario. Entonces si convertimos el número de marco en su versión hexadecimal, simplemente tendríamos que ponerlo como dígito más significativo en la dirección física si queremos dar el resultado en esta base.

a. Convertir las direcciones virtuales a direcciones físicas

Dirección virtual: 0x621C
Convertimos 0x621C a binario: 0110 | 0010 0001 1100

1. Número de página: $0110_2 = 6_{10}$
2. Desplazamiento: $0010\ 0001\ 1100_2 = 0x21C_{16}$
3. Consultamos la tabla de páginas: la página 6 está en el marco $8_{10} = 8_{16}$
4. Dirección física: **$0x821C_{16}$**

Dirección virtual: 0xF0A3
Convertimos 0xF0A3 a binario: 1111 | 0000 1010 0011
1. Número de página: $1111_2 = 15_{10}$
2. Desplazamiento: $0000\ 1010\ 0011_2 = 0x0A3_{16}$
3. Consultamos la tabla de páginas: la página 15 está en el marco $2_{10} = 2_{16}$

4. Dirección física: **0x20A3$_{16}$**

Dirección virtual: 0xBC1A
Convertimos 0xBC1A a binario: 1011 | 1100 0001 1010

1. Número de página: $1011_2 = 11_{10}$
2. Desplazamiento: 1100 0001 $1010_2 = 0xC1A_{16}$
3. Consultamos la tabla de páginas: la página 11 está en el marco $4_{10} = 4_{16}$
4. Dirección física: **0x4C1A$_{16}$**

Dirección virtual: 0x5BAA
Convertimos 0x5BAA a binario: 0101 | 1011 1010 1010

1. Número de página: $0101_2 = 5_{10}$
2. Desplazamiento: 1011 1010 $1010_2 = 0xBAA_{16}$
3. Consultamos la tabla de páginas: la página 5 está en el marco $13_{10} = D_{16}$
4. Dirección física: **0xDBAA$_{16}$**

Dirección virtual: 0x0BA1
Convertimos 0x0BA1 a binario: 0000 | 1011 1010 0001

1. Número de página: $0000_2 = 0_{10}$
2. Desplazamiento: 1011 1010 $0001_2 = 0xBA1_{16}$
3. Consultamos la tabla de páginas: la página 0 está en el marco $9_{10} = 9_{16}$
4. Dirección física: **0x9BA1$_{16}$**

Dirección Virtual	Número de Página	Desplazamiento	Marco	Dirección Física	Bit de Referencia
0x621C	6	0x21C	8	0x821C	1
0xF0A3	15	0x0A3	2	0x20A3	1
0xBC1A	11	0xC1A	4	0x4C1A	1
0x5BAA	5	0xBAA	13	0xDBAA	1
0x0BA1	0	0xBA1	9	0x9BA1	1

b. Dirección lógica que produce un fallo de página

Para conseguir esto, seleccionamos una dirección cuyo número de página ya esté en la tabla de páginas, pero que tenga un guión (-) en el campo de marco, lo que indica que la página no está actualmente cargada en memoria. Esto asegura que el sistema intentará buscar la página y encuentre que no está presente, causando así un fallo de página.

Podemos seleccionar, por ejemplo, la página 9 para este caso.

*Supongamos la dirección lógica **0x9A00***:

Convertimos **0x9A00** a binario: 1001 | 1010 0000 0000

1. Número de página: $1001_2 = 9_{10}$

2. Desplazamiento: 1010 0000 0000_2 = $0xA00_{16}$
3. Cuando intentamos acceder a la página 9, observamos en la tabla que tiene un guión (`-`), lo que indica que **no está en memoria**. Luego, como se indica en el quinto paso, se provoca un fallo de página ya que el sistema intentará acceder a esta página pero no podrá encontrarla en la memoria física.

c. Conjunto de marcos para LRU (*Least recently used*)

Utilizando el algoritmo de reemplazo de páginas (LRU) se reemplazará la página que ha estado en memoria sin ser usada por el mayor tiempo. En otras palabras, el sistema lleva un "historial" implícito de cuándo fue la última vez que cada página fue referenciada, y en caso de fallo de página, escoge la página que no ha sido referenciada en el período más largo.

En este caso, seleccionamos los marcos que tienen el bit de referencia en 0, ya que esto indica que esas páginas no han sido usadas recientemente.

Conjunto de marcos de página elegibles para el reemplazo: **{0, 1, 5, 6, 7, 10, 12, 15}**

Este conjunto incluye todos los marcos que actualmente contienen páginas cuyo bit de referencia está en 0. En caso de un fallo de página, el sistema seleccionará uno de estos marcos para reemplazar según el tiempo que cada página haya estado en ese estado inactivo.

Las páginas que no tienen un marco asignado (indicadas con un guión - en la tabla) no se consideran para el reemplazo, ya que no están cargadas en la memoria física en ese momento.

Ejecución de los programas

Cómo compilar y ejecutar los programas

Para ejecutar cualquier programa implementado a lo largo del proyecto, se han utilizado scripts .sh que facilitan el proceso de compilación y ejecución de cada una de las experiencias.

Lo único que se requiere es posicionarse en el directorio correspondiente al problema implementado y, una vez allí, ejecutar los siguientes comandos:

1. chmod +x run.sh
2. ./run.sh

```
e/proyecto-so/1.2. sincronizacion/1. taller-de-motos $ chmod +x run.sh
e/proyecto-so/1.2. sincronizacion/1. taller-de-motos $ ./run.sh
```

Ejemplo