UNS DCIC

SISTEMAS OPERATIVOS

PROYECTO

Comisión: 6
Santiago Maszong. LU: 125932
Cesar Semsey. LU: 106329

6 de Noviembre de 2024
Sistemas Operativos - Comisión N°3

Índice

1. Experimentación de Procesos y Threads con los Sistemas Operativos

1.1. Procesos, threads y Comunicación

1.1.1. Pumper Nic.

Las siguiente políticas e implementacones es común para ambos problema de la siguiente manera:

Procesos:

- **Empleado Despachador**: para este proceso se utilizó el proceso padre de todos los demás procesos, ya que es el que más interactúa con el resto.
  Este proceso, al ser el padre, es el creador e inicializador de los demás procesos.
  Luego de iniciar los demás procesos, empieza con su rol de despachador, que consta de las siguientes tareas en orden.

a. Chequear la cola de espera de entrada(dando prioridad a los vips).
b. Atiende a un cliente.
c. Crea la orden.
d. la despacha al empleado cocinero correspondiente.
e. vuelve al paso a

- **Clientes**: cada cliente en procesos hijos aparte. Cada cliente realiza la siguientes tareas en orden

a. Chequear si “Hay mucha gente esperando”, Si hay mucha gente vuelve a internet más tarde.
b. Si no “hay mucha gente esperando”, entonces decide que va comer(crea una orden) y espera a ser atendido por el empleado despachador.
c. Una vez atendido por el despachador, aguarda a que termine su orden.
d. Una vez que tiene su pedido se retira.

- **Empleado**: cada empleado “cocinero” en procesos hijos aparte, Empleado Hamburguesa simple, Empleado Menú Vegano, Empleado Papas fritas 1, Empleado Papas fritas 2. Cada empleado cocinero realiza las siguientes tareas en orden.

a. Chequear si llego una orden para el tipo de menú que prepara.
b. preparar el menú.
c. comunicar que la preparación fue terminada
d. volver al paso a

Se asume que:

- un menú/orden puede ser menú individualmente vegano, hamburguesa simple o papa fritas. Por ejemplo: no existe Hamburguesa simple con papas fritas para el menú.
- No está detallado como/quien comunica al cliente que su pedido finalizó, por comodidad optamos que sea el propio cocinero que comunique la finalización del pedido.

Para la implementación con pipes:

Se optó por usar los siguientes pipes:

- **un pipe por tipo de menú:** pipe_simple_burguer,pipe_vegan_menu, pipe_french_fries, utilizado para comunicar entre despachador y empleados cocineros, el despachador escribe la orden en el pipe correspondiente. Para el caso pipe_french_fries, dos procesos “Empleado Cocinero Papas Fritas” leen de este pipe.
- **un pipe para cada cola de entrada:** Se modeló el problema con una cola de entrada para cada tipo de cliente: pipe_entrance_queue, pipe_entrance_queue_vip. Los clientes según su tipo escribe su orden para el despachador en el correspondiente pipe.
- **un pipe por cliente:** pipe_order_completed[cliente_id], cada uno de estos pipes es para notificar al cliente que su orden ya se realizo.

Tareas de despachador:

- “Chequear la cola de entrada” se implementa usando espera ocupada. Primero leyendo de manera no bloqueante la cola de entrada vip(pipe_entrance_queue_vip) y en caso que no haya cliente vip, leer en la siguiente cola(pipe_entrance_queue). Se optó por modelar de esta manera para simular lectura de mensajes con prioridad, ya que no hay una manera por de leer por prioridad. Después las dos lecturas son no bloqueantes, ya que si alguna lo fuera podría haber inanición. Ej: queda bloqueado en la lectura de pipe_entrance_queue, y todos los clientes que se anuncian son vips, o viceversa.
- “Despachar la orden”: una vez que atendió al cliente, despacha la orden al empleado cocinero correspondiente escribiendo en el pipe correspiente del cocinero.

Tareas de Clientes:

- Chequear si “Hay mucha gente esperando”: El cliente obtiene la cantidad de clientes esperando en la cola correspondiente al cliente. Es decir, un cliente vip obtiene la cantidad de clientes de la cola de entrada vip, si esa cantidad sobrepasa una cierta cantidad, entonces hay mucha gente esperando y vuelve a intentarlo más tarde.
- “Espera a ser atendido por el empleado despachador”: el cliente crea su orden y la escribe en la cola correspondiente a su tipo. El cliente es atendido cuando el despachador lee la orden.
- “aguarda a que termine su orden”: El cliente queda a la espera de que llegue un mensaje a su pipe correspondiente(pipe_order_completed[cliente_id]) que notifique el pedido se realizo.

Tareas de Empleados Cocineros:

- “Chequear si llego una orden para el tipo de menú que prepara”: para esta implementación con pipe cada menú tiene su propio pipe, por lo tanto el cocinero queda a la espera de un mensaje en ese pipe.
- “comunicar que la preparación fue terminada”: escribe en el pipe correspondiente al cliente(pipe_order_completed[cliente_id]) que su orden se finalizó.

Para la implementación con Cola de Mensajes:

En todas las colas de mensajes, las estructura del mensajes tienen los mismos datos:

- priority: 1 si es vip, 2 sino
- client_id
- menu_id

Lo único que cambia es el orden de los campos, para poder usar el primer campo como tipo en los mensajes, según sea apropiado. Si bien algunos campos no cumplen una

funcionalidad en ciertas colas de mensajes o procesos, estos son usados para poder hacer una salida por consola amigable(ej: poder mostrar el menú en cualquier salida por pantalla).

Todos los envíos de mensajes son no bloqueantes y las recepción bloqueantes.

Se optó por usar las siguientes colas de mensajes:

- client_dispatcher_queue: esta cola es usada para comunicarse entre clientes(emisor) y despachador(receptor), simulará ser la cola de entrada. En esta cola el tipo de mensaje está dado por el campo priority, el cual es 1 para los clientes vip, 2 para los cliente comunes.
- dispatcher_employees_queue: esta cola es para la comunicación entre el despachador(emisor) y los cocineros(receptor). En esta cola el tipo de mensaje está dado por menu_id. Entonces
- order_finished_queue: Esta cola es usada para que el cocinero envie un mensaje al cliente correspondiente de que una orden se finalizó. En esta cola el tipo de mensaje está dado por client_id.

Tareas de despachador:

- "Chequear la cola de entrada": se implementa usando sin usar espera ocupada, a diferencia de con pipes. El despachador lee un mensaje de la cola de entrada(client_dispatcher_queue) con prioridad en si el cliente es vip. Por esa razón se optó que la estructura de mensaje de esta cola, el tipo de mensaje sea priority.
- "Despachar la orden": una vez que atendió al cliente, despacha la orden al empleado cocinero correspondiente enviado un mensaje con la orden y de tipo menu_id correspondiente.

Tareas de Clientes:

- Chequear si "Hay mucha gente esperando": El cliente obtiene la cantidad de clientes esperando en la cola de entrada,es decir la cantidad de mensajes en la cola de entrada(client_dispatcher_queue).
- "Espera a ser atendido por el empleado despachador": el cliente crea su orden y envía un mensaje de tipo priority correspondiente al cliente la cola de entrada(client_dispatcher_queue).
- "aguarda a que termine su orden": El cliente queda a la espera de que llegue un mensaje a su en la cola order_finished_queue, el cliente solo acepta un mensaje si el tipo mensaje es igual que su client_id.

Tareas de Empleados Cocineros:

- "Chequear si llego una orden para el tipo de menú que prepara": para esta implementación el cocinero solo lee mensajes de la cola dispatcher_employees_queue, sólo si el mensaje es del tipo igual al menu_id del menú que prepara el cocinero.
- "comunicar que la preparación fue terminada": el cocinero comunica que la orden finalizó enviando un mensaje de tipo cliente_id(la orden contiene el client_id correspondiente) a la cola order_finished_queue.

Ventajas de usar Cola de Mensajes:

Elimina la espera ocupa: en la implementación con pipes era necesario la espera ocupada para poder simular la cola de entrada con prioridad.

Economía de recursos: Son muchos menos las colas necesarias para la

comunicación entre procesos que la cantidad de pipes:

- comunicación entre clientes y dispatcher: 2 pipes vs 1 cola de mensajes.
- comunicación entre dispatcher y cocineros: 3 pipes vs 1 cola de mensajes.
- comunicación entre cocineros y clientes: 1 pipe por cliente vs 1 cola de mensajes.

Escalamiento: la implementación con pipes sufre el problema de necesitar 1 pipe por cliente para comunicar si un pedido fue finalizado. Entonces, al haber más clientes, se necesitarán más pipes. En cambio, con cola de mensajes, siempre será necesario 1 sola cola. Idem, si se quisiera agregar mas tipos de menú.

1.1.2. Mini Shell

Implementamos una minishell en un archivo llamado minishell.c.
Esta Mini Shell es un programa de línea de comandos simple que permite al usuario ejecutar un conjunto limitado de comandos relacionados con la gestión de directorios y archivos en un sistema. Al llamar a alguna de las funciones específicas de la minishell se crea un proceso hijo y con la función execv se cambia la imagen ejecutable del mismo por el archivo ejecutable de la función en cuestión. Todas las funciones de la minishell se encuentrar en archivos separados y al momento de correr el script de ejecución de la minishell estos archivos son compilados junto con el de la minishell, luego el script ejecuta el ejecutable de la minishell para comenzar con el programa.

- **ayuda()**:
  Esta función muestra una lista de comandos disponibles y una breve descripción de cada uno cuando se llama a "help" proporcionando información sobre cómo utilizar la Mini Shell.

- **crearDirectorio(char dir[])**:
  Esta función crea un directorio con el nombre especificado (dir) en la ubicación actual. Utiliza la función mkdir con permisos de lectura, escritura y ejecución para el propietario, el grupo y otros usuarios. Muestra un mensaje de éxito o un mensaje de error si la creación del directorio no es exitosa.

- **eliminarDirectorio(char dir[])**:
  Esta función elimina un directorio con el nombre especificado (dir) en la ubicación actual. Utiliza la función rmdir para eliminar el directorio.

- **crearArchivo(char file[])**:
  Esta función crea un archivo con el nombre especificado (file) en la ubicación actual. Utiliza la función fopen para crear un archivo en modo escritura ("w+"). Muestra en caso de éxito el correspondiente mensaje o un mensaje de error si la creación del archivo no es exitosa. Finaliza cerrando el archivo después de crearlo.

- **listarContenidoDirectorio(char dir[])**:
  Esta función lista el contenido de un directorio con el nombre especificado (dir) en la ubicación actual. Utiliza la función opendir para abrir el directorio y readdir para obtener y mostrar el contenido. Cierra el directorio después de listar el contenido.

- **mostrarContenidoArchivo(char dir[])**:
  Esta función muestra el contenido de un archivo con el nombre especificado (dir) en la ubicación actual. Utiliza la función fopen para abrir el archivo en modo lectura ("r") y luego muestra su contenido. Finaliza cerrando el archivo después de leerlo.
- **modificarPermisos(char dir[], int p)**:
  Esta función modifica los permisos de un archivo o directorio con el nombre especificado (dir). Utiliza la función chmod para cambiar los permisos al valor especificado por p.
  Esto permite cambiar los permisos de lectura, escritura y ejecución del archivo o directorio.
- **limpiar()**:
  Esta función se utiliza para limpiar el búfer de entrada después de leer una línea de comando.
  La función main muestra un mensaje de bienvenida y proporciona la opción "help" para obtener información sobre los comandos disponibles. Espera comandos del usuario. El usuario puede ingresar comandos como "mkdir", "rmdir", "mkfile", "ls", "showfile", "fileperm" ,"help" y "exit". Para cada comando, se realizan llamadas a las funciones correspondientes para llevar a cabo la operación deseada

1.2. Sincronización

1.2.1. Taller de Motos

a) Se utilizaron siete semáforos, uno para cada fase de construcción de una moto (construirNuevaMoto, rueda, chasis, motor, pintura, entregarMoto) y uno más para decidir cuál de los pintores pinta la moto (pintarMoto). Se definieron siete funciones, una para la funcionalidad de cada operario y una extra para manejar la entrega de la moto una vez terminada. Se usaron siete threads, uno para cada función representando a cada operario y a la funcionalidad de entrega de la moto para que se ejecuten concurrentemente. A continuación se explica cómo es la sincronización para obtener la secuencia de construcción de la moto.

Para obtener dicha secuencia de símbolos fue necesario inicializar el semáforo de construirNuevaMoto en 1 para desbloquear el hilo del operario de ruedas y poder comenzar con la construcción ya que todos los otros hilos están bloqueados.

Una vez desbloqueado el hilo de operario de ruedas el mismo “construye” en serie dos ruedas, aumentando el valor del semáforo ruedas en 2.

En este punto se desbloquea el hilo del operario del chasis, que necesitaba los 2 recursos del semáforo ruedas, y procede a armar el chasis aumentando el valor del semáforo chasis a 1 y desbloqueando al hilo del operario motor. Este, de manera análoga al anterior, coloca el motor y aumenta el valor del semáforo motor a 2 y el valor de pinturaMoto a 1.

Al aumentar el valor del semáforo del motor a 2 se desbloquean los dos hilos de los pintores (el que pinta de rojo y el que pinta de verde), para hacer la elección aleatoria de qué pintor realiza la pintura se utilizó la llamada no bloqueante try_wait sobre el semáforo pinturaMoto

en ambas funciones , sin mutex lock, de tal manera que se provoque una condición de carrera entre ambos para obtener el recurso. Finalmente ambos hilos se bloquean al esperar por el recurso del semáforo motor, sin el semáforo motor los hilos estarían haciendo busy waiting hasta que el try_wait en alguno de los 2 sea exitoso.

Cuando uno de las dos funciones accede al recurso del semáforo pinturaMoto procede a pintar la moto del color correspondiente y aumenta el recurso de los semáforos “pintura” y “entregarMoto” en 1 indicando que ya se realizó la misma.

El hilo que gestiona la entrega de las motos se desbloquea cuando consigue 2 recursos del semáforo entregarMoto, y como dicho semáforo se inicializa en 1, la primera moto es entregada luego de ser pintada (donde se agrega el segundo recurso necesario para desbloquear la entrega) omitiendo así el agregado del equipamiento extra en la primera moto. Para la segunda moto, luego de pintarla los valores de los semáforos “pintura” y “entregarMoto” son 2 y 1 respectivamente, aquí no se desbloquea la entrega de la moto todavía, sino que se desbloquea el último operario (el encargado del equipamiento extra) el cual requiere 2 recursos del semáforo “pintura”. Dicho operario agrega el equipamiento extra y aumenta en 2 los recursos del semáforo “entregarMoto”, el cual tiene ahora un valor de 3, permitiendo la entrega de la moto (consume 2) y dejando 1 recurso para que se repita el ciclo indefinidamente. De esta forma se obtiene que 1 de cada 2 motos se le agrega equipamiento extra alternadamente.

Cabe destacar que cada uno de las funciones que ejecutan los hilos mencionados tienen un ciclo while que vuelve a ejecutar el mismo código indefinidamente, con la particularidad de que se bloquea el hilo ya que la primer instrucción dentro del ciclo while es una llamada wait por un recurso de un semáforo entonces se bloquea hasta obtener dicho recurso como se explicó previamente, sin hacer busy waiting.

1.2.2. Santa Claus

Primero se definieron constantes arbitrarias para la cantidad de renos (9) y para la cantidad de elfos (6).
Se crearon 3 funciones, una para el comportamiento de santa, para los renos y para los elfos. Además se utilizó un hilo para cada entidad, uno para cada reno (9), uno para cada elfo (6) y uno para santa, donde cada uno de estos hilos se dedica a correr la función correspondiente con el comportamiento de la entidad que representa.

Santa: Santa duerme (se bloquea el hilo) esperando a que vengan los 9 renos al polo norte o 3 elfos a pedir ayuda (o ambos). Santa se despierta en alguno de estos dos casos cuando el semáforo ”elfosOrenos” tiene algún recurso disponible.

Santa no sabe si lo despertaron los renos listos para armar el trineo o los elfos para pedirle ayuda entonces para darle prioridad a los renos sobre los elfos si ambos necesitan a Santa, el mismo primero chequea si los renos lo despertaron haciendo un try_wait sobre el semáforo “renosListos”, y luego chequea si los elfos necesitan ayuda también con un try_wait pero sobre el semáforo “elfosConProblemas”.

Son necesarios los try_waits para que, en el caso de que no estén los renos listos o los elfos pidiendo ayuda, Santa no se bloquee esperando por dicho evento en específico potencialmente ignorando así el otro evento.

Si los renos están listos entonces el try_wait sobre “renosListos” da verdadero, Santa arma el trineo y se va con los renos para luego desbloquear a los mismos aumentando el valor del semáforo “armarTrineo” en 9 (por la cantidad de renos).

Si los renos no están listos entonces no despertaron a Santa y por ende los elfos lo hicieron, en tal caso el try_wait sobre “renosListos” da falso y Santa pasa directamente a darle atención a los elfos donde el try_wait sobre “elfosConProblemas” da verdadero y luego de ayudarlos desbloquea a los mismos aumentando el valor del semáforo “santaAyudaElfo” en 3 (ya que los elfos van de a 3 a pedir ayuda).

En el caso de que ambos despierten a Santa, se atiende primero a los renos y luego a los elfos.

Luego de chequear por los renos y los elfos, y atender los eventos necesarios Santa vuelve a dormirse (bloquearse) esperando por un recurso del semáforo “elfoaOrenos” nuevamente.

Los Renos: Cada uno de los renos pasa un tiempo aleatorio (entre 10s y 20s) en el trópico antes de volver al polo norte y esperar por los demás renos, consumiendo un recurso del semáforo “renosEnPoloNorte”, que tiene un valor inicial de 9 (por el número de renos).

Este semáforo es necesario para identificar al último de los 9 renos en llegar al polo norte de la siguiente manera:

Cada reno al llegar al polo norte hace un try_wait sobre el semáforo “renosEnPoloNorte”, si da verdadero es que el semáforo tiene recursos, por ende quedan más renos por volver del trópico y no es el último. Aquí el reno aumenta en 1 el valor de dicho semáforo ya que al hacer el try_wait y dar verdadero se consumió un recurso en vano. Luego el reno se bloquea esperando por el semáforo “armarTrineo”.

Si el reno es el último de los 9 en volver, el try_wait sobre “renosEnPoloNorte” va a dar falso ya que se habrán consumido sus 9 recursos y es allí donde este último reno despierta a Santa aumentando el valor del semáforo “elfosOrenos” en 1 e indicando que fueron los renos los que pidieron su atención aumentando el valor del semáforo “renosListos” en 1.

Santa procede a atenderlos, armar el trineo, viajar y luego libera a los renos aumentando el valor del semáforo “armarTrineo” en 9 (por la cantidad de renos).

Cada uno de los renos devuelve el recurso tomado del semáforo “renosEnPoloNorte” indicando que se van del polo norte y dejando al semáforo con valor 9 como en un inicio. A

partir de aquí se repite el ciclo.

Cabe destacar que para evitar condiciones de carrera a la hora de hacer try_wait sobre “renosEnPoloNorte” entre los renos se utilizó un mutex lock.

Los Elfos: Los elfos comienzan pasando un tiempo aleatorio trabajando (entre 5s y 10s) antes de tener un problema y necesitar ayuda de Santa consumiendo un recurso del semáforo “elfosParaDespertarASanta”, que tiene un valor inicial de 3 ya que son los necesarios para despertar a Santa.

Cada elfo al necesitar ayuda de Santa hace un try_wait sobre el semáforo “elfosParaDespertarASanta”, si da verdadero es que el semáforo tiene recursos, por ende todavía no se obtuvieron los 3 elfos necesarios para ir a despertar a Santa. En este caso el elfo aumenta en 1 el valor de dicho semáforo ya que al hacer el try_wait y dar verdadero se consumió un recurso en vano. Luego el elfo se bloquea esperando por el semáforo “santaAyudaElfo”.

Si el elfo es el tercero en necesitar ayuda, el try_wait sobre “elfosParaDespertarASanta” va a dar falso ya que se habrán consumido sus 3 recursos y es allí donde este último elfo despierta a Santa aumentando el valor del semáforo “elfosOrenos” en 1 e indicando que fueron los elfos los que pidieron su atención aumentando el valor del semáforo “elfosConProblemas” en 1.

Santa procede a atenderlos, ayudarlos con sus problemas y luego los elfos se van aumentando el valor del semáforo “santaAyudaElfo” en 3 (por la cantidad de elfos atendidos).

Los 3 elfos devuelven el recurso tomado del semáforo “elfosParaDespertarASanta” indicando que se vuelven a trabajar y dejando al semáforo con valor 3 como en un inicio. A partir de aquí se repite el ciclo.

Si durante todo este proceso algún elfo extra necesita ayuda de Santa el mismo se va a quedar bloqueado esperando por un recurso del semáforo “elfosParaDespertarASanta”, es decir que tiene esperar a que los 3 elfos que ya están esperando ayuda de Santa o se están atendiendo con él terminen y vuelvan a trabajar. Además tiene que esperar a que otros 2 elfos tengan problemas para ser un total de 3 y poder pedir ayuda a Santa.

Cabe destacar que para evitar condiciones de carrera a la hora de hacer try_wait sobre “elfosParaDespertarASanta” entre los elfos se utilizó un mutex lock. En la exclusión mutua también se incluyó el wait sobre el semáforo “elfosParaDespertarASanta” ya que podría suceder que si varios elfos hacen wait sobre dicho semáforo casi a la vez agotándose los recursos del mismo la consulta sobre el try_wait podría dar false para más de un elfo, agregando así recursos extras a los semáforos “elfosOrenos” y “elfosConProblemas” y rompiendo la sincronización.

2 Problemas Conceptuales

2.1 Paginación y Segmentación

a) La dirección lógica se compone por los bits de número de página (más significativos) y los bits de desplazamiento dentro de la página (menos significativos). Como la dirección es de 16 bits y el tamaño de página es de 512 direcciones, lo que es equivalente a 2^9 direcciones, entonces son necesarios 9 bits para direccionar todas las direcciones de una página. Por lo que los bits de desplazamiento dentro de la página son los 9 menos significativos. Eso nos deja los 7 bits más significativos para el número de página.
Entonces la dirección 0011000000110011 separada queda 0011000 (7 bits de páginas) 000110011 (9 bits de desplazamiento).
Luego, como el número de marco M es M = P/2, con P siendo el número de página y en nuestro caso el número de página es 0011000_2 (24_{10}) entonces el número de marco M = $24/2 = 12_{10} = 1100_2$.
Finalmente para obtener la dirección física final se concatena el número de marco obtenido con los bits de desplazamiento originales obteniéndose la dirección física
001100 000110011

b) Como el tamaño máximo de segmento es de 2K direcciones son necesarios 11 bits para direccionar el desplazamiento en un segmento ya que 2^11 = 2084 > 2K direcciones.
Entonces los 11 bits menos significativos de la dirección lógica indican el desplazamiento dentro del segmento.
Finalmente los 5 bits restantes de los 16 bits son para número de segmento y la dirección 0011000000110011 separada queda 00110 (5 bits de segmento) 00000110011 (11 bits de desplazamiento).
Como el número de segmento es $00110_2 = 6_{10}$ y el desplazamiento es $00000110011_2 = 51_{10}$ se puede calcular la base como Base = $20_{10} + 4096_{10} + 6_{10} = 4122$, y por último se suma la Base con el desplazamiento obteniendo la dirección física: $4122_{10} + 51_{10} = 4173_{10}$.

2.2 Paginación con LRU Local

Las direcciones lógicas y físicas de 16 bits se componen por los bits de número de página/marco (más significativos) y los bits de desplazamiento dentro de la página (menos significativos). Ya que las páginas tienen un tamaño de 4096 bytes, equivalente a 2^12 bytes, y como el direccionamiento es al byte entonces son necesarios 12 bits para direccionar el desplazamiento dentro de cada página. Entonces de los 16 bits totales de la dirección los 12 menos significativos son los de desplazamiento y los 4 más significativos son los de numero de página/marco. Esto se corrobora al ver que la tabla de páginas tiene 16 entradas o páginas y que dichos bits se utilizan para navegar en la tabla.

a) Para obtener las direcciones físicas equivalentes debo tomar los primeros 4 bits de la dirección lógica ya que los mismos indican el número de página.
Para el caso de la dirección 0x621C, que en binario es 0110 0010 0001 1100, los 4 bits más significativos representan al número 6. Localizo dicha página en la tabla y obtengo el número de marco que es 8. Este número de marco se lo concateno a los 12 bits restantes de la dirección lógica (bits de desplazamiento) para obtener la dirección física final de 0x821C.

Dirección Virtual	Número de Página	Marco	Desplazamiento	Dirección Física
0x621C	6	8	0x21C	0x821C
0xF0A3	15 (F_{16})	2	0x0A3	0x20A3
0xBC1A	11 (B_{16})	4	0xC1A	0x4C1A
0x5BAA	5	13 (D_{16})	0xBAA	0xDBAA
0x0BA1	0	9	0xBA1	0x9BA1

Así obtengo todas las direcciones físicas y los bits de referencia para dichas páginas se setea en 1, indicando que se ha accedido recientemente a los marcos de la memoria física indicados en la tabla.

La tabla queda así:

Pagina	Marco	Bit 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) Una dirección lógica que produzca un fallo de página es, por ejemplo la dirección 0x1A23 que representa a la entrada 1 en la tabla de páginas y cuyo referenciamiento a un marco en memoria principal no está definido ya que muestra un guión (-).

c) En este caso, si hay lugar en memoria principal se puede cargar dicha página en un marco en la misma y referenciar su dirección en un hueco libre de la tabla de páginas. Sino, como el reemplazo es local, el proceso que causa el fallo de página deberá reemplazar una página de un marco perteneciente a su propio conjunto de marcos asignados. Y como el algoritmo de reemplazo es LRU la página a reemplazar debe ser una no recientemente usada. Dicha página se puede obtener observando el bit de referencia, y reemplazarla por la nueva página que necesita cargarse en memoria.

En definitiva el marco de página seleccionado para realizar el reemplazo va a ser alguno del conjunto de marcos asignados al proceso en cuestión y además que tenga el bit de referencia en 0.