



Sea un espacio de direcciones lógico de 1024 páginas de 8 bytes cada una, sobre una memoria física (DRAM) de tan sólo 64 bytes direccionable a nivel de byte en la que conviven 3 procesos de 32 bytes cada uno, A, B y C, representados respectivamente por las direcciones A0...A31, B0...B31, C0...C31. Considera una memoria DRAM inicialmente vacía que utiliza el algoritmo FIFO para el reemplazo de páginas. Se pide determinar las direcciones de memoria virtual alojadas en las 64 posiciones de memoria física cuando se han solicitado determinadas secuencias de direcciones que se indican a continuación:

1. Para la secuencia de direcciones lógicas {A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24}, la memoria física contendrá los valores correspondientes a las direcciones

- ☐ a) A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24, y el resto de posiciones vacías.
- ☐ b) B0, B8, B16, B24, C0, C8, C16, C24, y el resto de posiciones quedan vacías.
- ☐ c) C0 a C31 (proceso C completo) seguido de B0 a B31 (proceso B completo), y no queda ninguna posición vacía.
- ☐ d) Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☒ c). Cada acceso a memoria virtual mete en memoria física la dirección solicitada y las 7 que le acompañan en su misma página. Así, comienzan entrando en memoria física las 4 páginas del proceso A, seguido de las 4 páginas del proceso B, momento en que se llenan las 8 páginas de memoria física disponibles. Al referenciarse a continuación las 4 páginas del proceso C, éstas van reemplazando una a una a las 4 páginas del proceso A, pues habían sido las primeras en alojarse y son las elegidas por el algoritmo FIFO. De esta manera, las 4 páginas de B quedan ubicadas en la segunda mitad de la de memoria física, y las 4 páginas de C terminan ocupando la primera mitad, usurpando las posiciones que inicialmente ocuparon las páginas del proceso A.

2. Para la secuencia de direcciones lógicas {A0, A4, A8, A12, A16, A20, A24, A28, B0, B4, B8, B12, B16, B20, B24, B28, C0, C4, C8, C12, C16, C20, C24, C28}, la memoria física contendrá los valores correspondientes a las direcciones

- ☐ a) A0, A4, A8, A12, A16, A20, A24, A28, B0, B4, B8, B12, B16, B20, B24, B28, C0, C4, C8, C12, C16, C20, C24, C28, y el resto de posiciones quedan vacías.
- ☐ b) B0, B4, B8, B12, B16, B20, B24, B28, C0, C4, C8, C12, C16, C20, C24, C28, y el resto de posiciones quedan vacías.
- ☐ c) C0 a C31 (proceso C completo) seguido de B0 a B31 (proceso B completo), y no queda ninguna posición vacía.
- ☐ d) Ninguna de las respuestas anteriores es correcta.


Respuesta correcta: ☒ c). Dado que cada acceso a memoria virtual arrastra a memoria física su palabra y las de las 7 direcciones que le acompañan en su misma página virtual, la secuencia de páginas referenciada coincide con la de la cuestión anterior, por lo que su alojamiento en memoria física es también el mismo.

3. Para la secuencia de direcciones lógicas {A0, B0, C0, A1, B1, C1, A2, B2, C2, y así proseguimos con todos los números ordenadamente hasta concluir con A31, B31, C31}, quedarán en memoria física los valores correspondientes a

- a** Las primeras 64 direcciones solicitadas de ese total de 96 direcciones.
- b** Las últimas 64 direcciones solicitadas de ese total de 96 direcciones.
- c** Las dos últimas páginas del proceso A, las tres últimas páginas del proceso B y las tres últimas páginas del proceso C.
- d** Todas las páginas de los procesos B y C, y ninguna del proceso A.

Respuesta correcta: **c**. Según avanza la secuencia de peticiones, la primera página de A, B y C son las tres primeras en ser reemplazadas cuando nos quedamos sin memoria física, ya que el algoritmo FIFO escoge las primeras que entraron (la primera página de A contiene las direcciones A0 a A7, la primera página de B contiene las direcciones B0 a B7 y la primera página de C contiene las direcciones C0 a C7). Dado que la secuencia de direcciones solicitada se expande a lo largo de 12 páginas (4 por cada uno de los 3 procesos), es necesario sacrificar una página más para meter las últimas direcciones, y ésta será la segunda página del proceso A (que por FIFO es el proceso más madrugador en introducir su segunda página en memoria).

Sugerencia: A partir de este ejercicio, las cuestiones se van complicando y cuesta llevar un seguimiento a la secuencia de eventos. Para ello, recomendamos utilizar los naipes de una baraja para representar a las direcciones de memoria. Inicialmente, se puede usar un naipe para cada dirección solicitada, siendo necesarias tres barajas para simbolizar los tres procesos A, B y C. Una vez que el alumno se haya familiarizado con el concepto de página y tenga claro que cada una de ellas agrupa 8 direcciones de memoria, puede simplificar el modelo considerando un naipe para cada página de memoria referenciada, y de esta manera, cada proceso vendría representado por un palo de la baraja (oros para el proceso A, copas para el B y espadas para el C, por ejemplo). A partir de ahí, pueden desplegarse los naipes sobre una mesa siguiendo las secuencias de direcciones dentro de un proceso en vertical, y entre procesos en horizontal, de manera que cada tipo de recorrido coincida con el ejercicio concreto (los naipes se irían colocando de arriba a abajo para simular las secuencias de los dos primeros ejercicios, y de izquierda a derecha para replicar la secuencia este tercer ejercicio). Cuando una página se reemplaza, se le da la vuelta a su naipe para reflejar que ha pasado a disco, y durante este juego hay que tener claro que sólo pueden estar boca arriba sobre la mesa un máximo de 8 naipes, representando a las 8 páginas que caben en memoria física. De esta manera, levantando y ocultando los naipes podemos hacer un seguimiento de las páginas que entran y salen de DRAM, efectuando sobre la marcha un conteo del número de faltas de página que se van sucediendo. El siguiente diagrama ilustra la disposición sugerida de los naipes sobre la mesa, así como las secuencias de direcciones solicitadas en las primeras tres cuestiones y las páginas a las que pertenecen dentro de nuestra representación.

Direcciones de cada proceso agrupadas en sus 4 páginas:			Secuencia de dirs. para la cuestión 1:			Secuencia de dirs. para la cuestión 2:			Secuencia de dirs. para la cuestión 3:			Sugerencia para representar y manipular las páginas en cuestiones sucesivas:		
A0 A1 A2 A3 A4 A5 A6 A7	B0 B1 B2 B3 B4 B5 B6 B7	C0 C1 C2 C3 C4 C5 C6 C7	A0 A1 A2 A3 A4 A5 A6 A7	B0 B1 B2 B3 B4 B5 B6 B7	C0 C1 C2 C3 C4 C5 C6 C7	A0 A1 A2 A3 A4 A5 A6 A7	B0 B1 B2 B3 B4 B5 B6 B7	C0 C1 C2 C3 C4 C5 C6 C7	A0 A1 A2 A3 A4 A5 A6 A7	B0 B1 B2 B3 B4 B5 B6 B7	C0 C1 C2 C3 C4 C5 C6 C7			
A8 A9 A10 A11 A12 A13 A14 A15	B8 B9 B10 B11 B12 B13 B14 B15	C8 C9 C10 C11 C12 C13 C14 C15	A8 A9 A10 A11 A12 A13 A14 A15	B8 B9 B10 B11 B12 B13 B14 B15	C8 C9 C10 C11 C12 C13 C14 C15	A8 A9 A10 A11 A12 A13 A14 A15	B8 B9 B10 B11 B12 B13 B14 B15	C8 C9 C10 C11 C12 C13 C14 C15	A8 A9 A10 A11 A12 A13 A14 A15	B8 B9 B10 B11 B12 B13 B14 B15	C8 C9 C10 C11 C12 C13 C14 C15			
A16 A17 A18 A19 A20 A21 A22 A23	B16 B17 B18 B19 B20 B21 B22 B23	C16 C17 C18 C19 C20 C21 C22 C23	A16 A17 A18 A19 A20 A21 A22 A23	B16 B17 B18 B19 B20 B21 B22 B23	C16 C17 C18 C19 C20 C21 C22 C23	A16 A17 A18 A19 A20 A21 A22 A23	B16 B17 B18 B19 B20 B21 B22 B23	C16 C17 C18 C19 C20 C21 C22 C23	A16 A17 A18 A19 A20 A21 A22 A23	B16 B17 B18 B19 B20 B21 B22 B23	C16 C17 C18 C19 C20 C21 C22 C23			
A24 A25 A26 A27 A28 A29 A30 A31	B24 B25 B26 B27 B28 B29 B30 B31	C24 C25 C26 C27 C28 C29 C30 C31	A24 A25 A26 A27 A28 A29 A30 A31	B24 B25 B26 B27 B28 B29 B30 B31	C24 C25 C26 C27 C28 C29 C30 C31	A24 A25 A26 A27 A28 A29 A30 A31	B24 B25 B26 B27 B28 B29 B30 B31	C24 C25 C26 C27 C28 C29 C30 C31	A24 A25 A26 A27 A28 A29 A30 A31	B24 B25 B26 B27 B28 B29 B30 B31	C24 C25 C26 C27 C28 C29 C30 C31			

Este rectángulo simboliza una página, y dentro escribimos las direcciones que corresponden a las 8 palabras que alberga en memoria. Igualmente pueden ser representadas por un naipe sobre una mesa para que nos permita replicar el movimiento de datos que se produce durante la gestión de páginas por parte del Sistema Operativo

4. Para la secuencia de direcciones lógicas anterior, pero recorrida en el orden inverso, es decir, comenzando por la última y finalizando por la primera, quedarán en memoria física los valores correspondientes a

- ☐ a Las primeras 64 direcciones solicitadas de ese total de 96 direcciones.
- ☐ b Las últimas 64 direcciones solicitadas de ese total de 96 direcciones.
- ☐ c Las tres primeras páginas del proceso A, las tres primeras páginas del proceso B y las dos primeras páginas del proceso C.
- ☐ d Todas las páginas de los procesos A y B, y ninguna del proceso C.

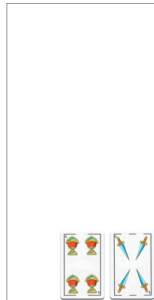
Respuesta correcta: ☒ c. El proceso es equivalente al descrito en la cuestión anterior, sólo que cambiando la numeración de páginas en el orden decreciente. Podemos verlo mejor si hacemos un seguimiento a la secuencia de peticiones con los naipes sobre la mesa, para lo cual ilustraremos únicamente las 12 peticiones que producen una falta de página (y percibiremos el consiguiente reemplazo de página cuando veamos voltear el naipe que la representa). La secuencia de peticiones es la siguiente:

C31, B31, A31, ..., C23, B23, A23, ... , C15, B15, A15, ... , C7, B7, A7, ... , hasta acabar con A0
La composición de la mesa de naipes después de cada una de las 12 peticiones queda como sigue (el resto de peticiones no alteran la mesa porque se benefician del tamaño de página, y son cargadas en memoria junto con una petición anterior que está en su misma página):

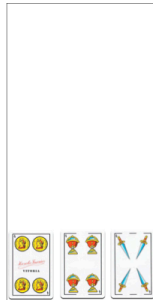
Mesa tras C31:



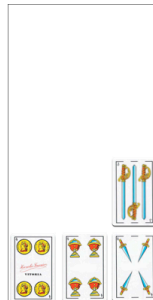
Mesa tras B31:



Mesa tras A31:



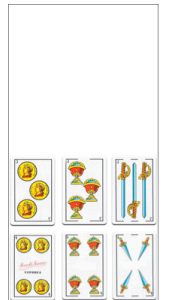
Mesa tras C23:



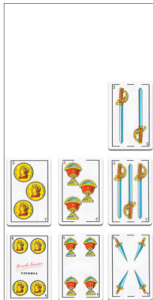
Mesa tras B23:



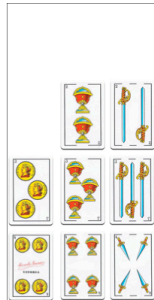
Mesa tras A23:



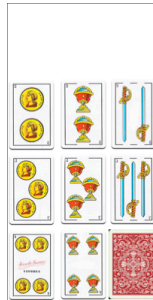
Mesa tras C15:



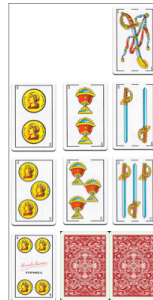
Mesa tras B15:



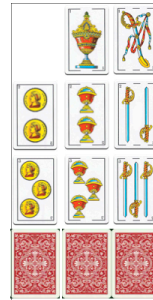
Mesa tras A15:



Mesa tras C7:



Mesa tras B7:



Mesa tras A7:



5. Si cambiáramos el algoritmo de reemplazo por LRU en lugar de FIFO, ¿Cambiaría el contenido final de la memoria en alguna de las dos secuencias anteriores?

- ☐ a Sí, cambiaría para la primera secuencia (la creciente), pero no para la segunda (la inversa).
- ☐ b Sí, cambiaría para la segunda secuencia (la inversa), pero no para la primera (la creciente).
- ☐ c No cambiaría para ninguna de las dos secuencias.
- ☐ d Cambiaría para las dos secuencias.

Respuesta correcta: **[c]**. En nuestro caso, las páginas que van entrando de izquierda a derecha en la mesa de naipes, también van siendo sucesivamente barridas en ese orden, por lo que la menos recientemente referenciada será siempre la que vaya estando más a la izquierda de la fila más inferior de naipes que tenga páginas en memoria física, que también es la primera que entró por FIFO. Por lo tanto, las 4 últimas páginas que entran, que son las que reemplazan a otras ya existentes, seleccionarán las mismas víctimas en ambos algoritmos (ver mesas del ejercicio anterior):

- El dos de oros saca al cuatro de espadas.
- El as de espadas saca al cuatro de copas.
- El as de copas saca al cuatro de oros.
- El as de oros saca al tres de espadas.

6. Sea la secuencia de direcciones lógicas anterior, $\{A0, B0, C0, A1, B1, C1, A2, B2, C2, \dots\}$, seguida inmediatamente por la secuencia inversa $\{C31, B31, A31, C30, B30, A30, C29, B29, A29, \dots$ hasta regresar a $A0\}$. Si seguimos utilizando el algoritmo FIFO para el reemplazo, quedarán en memoria física los valores correspondientes a

- [a]** Las últimas 64 direcciones solicitadas de ese total de 96 direcciones.
- [b]** Las tres primeras páginas del proceso A, las tres primeras páginas del proceso B y las dos primeras páginas del proceso C.
- [c]** Todas las páginas de los procesos A y B, y ninguna del proceso C.
- [d]** Las dos primeras páginas de los procesos A, B y C, y la última página de los procesos B y C.

Respuesta correcta: **[d]**. Sea $P\#X$ la página $\#$ del proceso X . Al iniciar la secuencia inversa de peticiones, el contenido de la memoria tiene las 8 páginas siguientes de más a menos antigua (comenzando a numerar en 0): $P1B, P1C, P2A, P2B, P2C, P3A, P3B, P3C$. A partir de ahí:

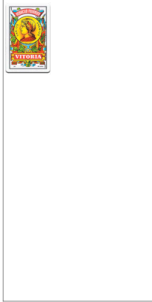
- $P1A$ reemplaza a $P1B$.
- $P1B$ reemplaza a $P1C$.
- $P1C$ reemplaza a $P2A$.
- $P0C$ reemplaza a $P2B$.
- $P0B$ reemplaza a $P2C$.
- $P0A$ reemplaza a $P3A$.

$P1B$ y $P1C$ son reemplazadas y posteriormente repescadas durante la secuencia. Quedan en memoria física 8 páginas del total de 12 páginas lógicas referenciadas, y alojadas en disco las últimas 4 reemplazadas, esto es, $P2A, P2B, P2C$ y $P3A$.

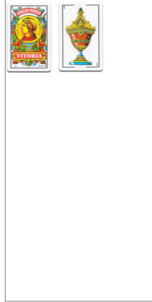
Si hemos entendido la representación con naipes, todo es más sencillo. Coloquemos la secuencia de peticiones mencionando, únicamente, las peticiones a memoria que provocan una falta de página, e iremos volteando naipes según se vayan sucediendo los correspondientes reemplazos:

$A0, B0, C0, A8, B8, C8, A16, B16, C16, A24, B24, C24, A15, B14, C13, C7, B7, A7$

Mesa tras A0:



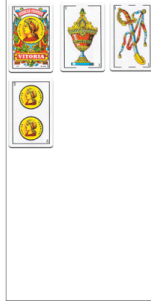
Mesa tras B0:



Mesa tras C0:



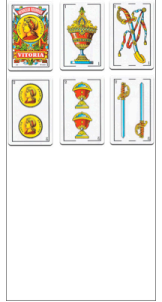
Mesa tras A8:



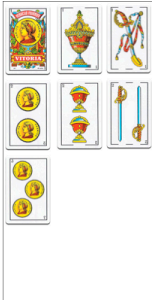
Mesa tras B8:



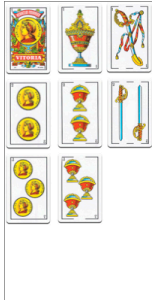
Mesa tras C8:



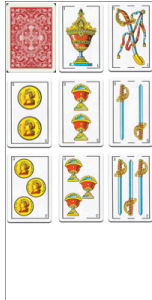
Mesa tras A16:



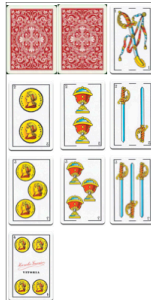
Mesa tras B16:



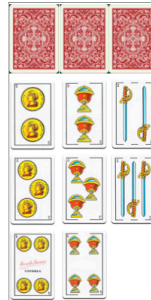
Mesa tras C16:



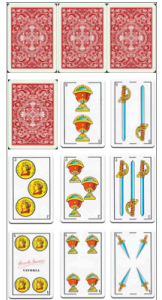
Mesa tras A24:



Mesa tras B24:



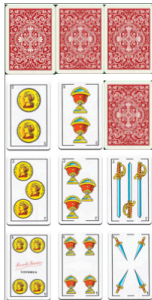
Mesa tras C24:



Mesa tras A15:



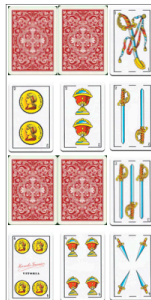
Mesa tras B14:



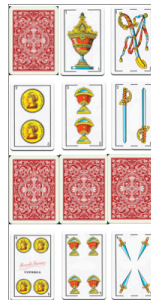
Mesa tras C13:



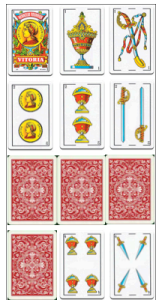
Mesa tras C7:



Mesa tras B7:



Mesa tras A7:



7. ¿Cuántas faltas de página se producen en total al tramitar la secuencia completa de peticiones anterior? (es decir, numeración ascendente seguida de numeración descendente). Recuerda que debes considerar como falta de página tanto la llegada inicial de una página a la memoria vacía como su posterior reemplazo por otra.

- ☐ a 13.
☐ b 16.
☐ c 18.
☐ d 20.

Respuesta correcta: ☒ c. En la secuencia creciente, se referencian 12 páginas, dando lugar a esas mismas faltas de página puesto que se parte de una memoria vacía. En la secuencia inversa se producen 6 faltas más según se ha indicado en la solución de la cuestión anterior. En total, son $12+6 = 18$ faltas de página.

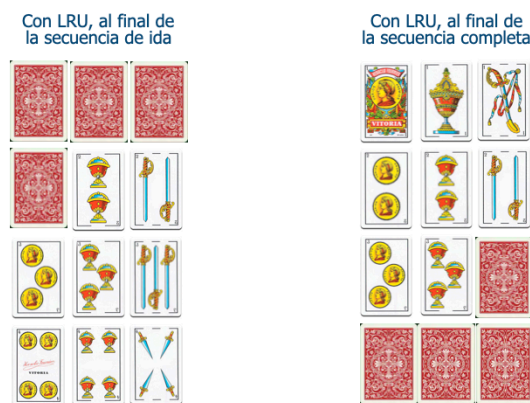
8. Si usamos el algoritmo de reemplazo LRU en lugar de FIFO en la secuencia de peticiones anterior (es decir, numeración ascendente seguida de numeración descendente), ¿Cómo cambiaría el contenido final de la memoria?

- ☐ a Cambiaría durante la primera mitad de la secuencia (esto es, numeración ascendente), pero no para la segunda mitad (numeración descendente).
- ☐ b Cambiaría durante la segunda mitad de la secuencia (esto es, numeración descendente), pero no para la primera mitad (numeración ascendente).
- ☐ c No cambiaría para ninguna de las dos secuencias.
- ☐ d Cambiaría para las dos secuencias.

Respuesta correcta: ☒ b. El tramo de numeración ascendente acaba con las últimas 8 páginas referenciadas en ambos casos, puesto que también son las últimas 8 introducidas en DRAM. En el último tramo de numeración descendente, LRU irá sacrificando las páginas P3C, P3B, P3A y P2C (por este orden), ya que van quedando sucesivamente más lejos en el tiempo desde la última vez que se referenciaron. Esto no coincide con los reemplazos que hemos visto anteriormente para el algoritmo FIFO, ya que son las últimas páginas que se introdujeron en memoria, y FIFO elige las primeras. Si utilizamos los naipes, al final de la secuencia de ida, el estado es el que se muestra abajo a la izquierda, que en este caso coincide con el correspondiente al algoritmo de reemplazo FIFO. Pero en la secuencia inversa, los reemplazos son bien distintos, ya que LRU va a comenzar a descartar los naipes que están más abajo y a la derecha en la matriz de 4 filas y 3 columnas. Los reemplazos durante la secuencia inversa utilizando el algoritmo LRU serían los siguientes:

- El dos de oros reemplaza al cuatro de espadas.
- El as de espadas reemplaza al cuatro de copas.
- El as de copas reemplaza al cuatro de oros.
- El as de oros reemplaza al tres de espadas.

Por lo tanto, se producen dos faltas de página menos que cuando reemplazamos con FIFO, y la situación final sería la que se muestra abajo a la derecha.



9. ¿Cuántas faltas de página se producen en total al tramitar la secuencia completa de peticiones anterior? (es decir, numeración ascendente seguida de numeración descendente). Recuerda que debes considerar tanto la llegada inicial de una página a la memoria vacía como su posterior reemplazo por otra.

- ☐ a 13.
- ☐ b 16.
- ☐ c 18.

☐ d 20.

Respuesta correcta: ☐ b. En la secuencia de ida (la creciente), se referencian 12 páginas, dando lugar a esas mismas faltas de página puesto que se parte de una memoria vacía. En la secuencia inversa (la decreciente) se producen 4 faltas más según se ha indicado en la solución de la cuestión anterior. En total, son $12+4 = 16$ faltas de página. Frente a FIFO, la secuencia de numeración decreciente no selecciona para reemplazar ninguna de las páginas P1A, P1B o P1C, y por tanto, nos ahorramos los dos reemplazos necesarios en FIFO para la posterior repesca de P1B y P1C (esto es, el dos de copas y el dos de espadas).

10. Diseña una secuencia de peticiones de memoria que finalice dejando en memoria física todas las direcciones del proceso A que son múltiplo de 5 y fuera de ella todas las direcciones del proceso A que son múltiplo de 7.

☐ a A5, A10, A15, A20, A25, A30.

☐ b A0, A1, A2, A3, A4, A5, A6, A8, A9, A10, A11, A12, A13, A15, A16, A17, A18, A19, A20, A22, A23, A24, A25, A26, A27, A29, A30, A31.

☐ c Las dos respuestas anteriores son correctas.

☐ d Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☐ d. Al convivir en una misma página de memoria las direcciones A5 y A7, que son múltiplos de 5 y 7 respectivamente, el hecho de que no podamos partir una página impide cumplir con los requisitos señalados en la cuestión: Nunca podremos diseñar una secuencia que incluya A5 y no incluya A7, puesto que están en la misma página.

11. Indicar la longitud de los campos p, f y d con los que se componen las direcciones de memoria lógica (campo p seguido de d) y física (campo f seguido de d).

☐ a p=10, f=3, d=3.

☐ b p=10, f=6, d=0.

☐ c p=10, f=6, d=3.

☐ d p=13, f=6, d=3.

Respuesta correcta: ☐ a.

- Para calcular p: Hay 1024 páginas lógicas, esto es, 2^{10} páginas. Dado que p es su logaritmo binario, p=10.
- Para calcular f: La memoria física es de 64 bytes, y como las páginas son de 8 bytes, tenemos 8 páginas de memoria física (de 8 bytes cada una). f es el logaritmo binario del número de páginas de memoria física (o marcos de memoria), esto es, $8 = 2^3$ para darnos f=3.
- Para calcular d: Las páginas son de 8 bytes con palabras de 1 byte (la unidad direccionable de la memoria), por lo que cada página tiene 8 palabras. El campo desplazamiento, d, direcciona la palabra dentro de la página, y por lo tanto, su longitud es el logaritmo binario del número de palabras por página. Puesto que hay 8 palabras por página, $8 = 2^3$, resultando d=3.

12. De los campos anteriores, ¿cuáles estarían dentro de la TLB?

☐ a p, f y d.

☐ b p y f.

☐ c f y d.

☐ d p y d.

Respuesta correcta: ☒ b. La TLB (Translation Look-Aside Buffer) es una memoria caché que contiene las traducciones de direcciones lógicas a físicas más utilizadas, y por lo tanto, cada entrada en la TLB albergará una pareja compuesta por la dirección lógica y su traducción a dirección física o *frame* (marco).

13. ¿Reduciría la presencia de la TLB el número de faltas de página en la secuencia de peticiones creciente seguida de la secuencia decreciente que hemos visto anteriormente?

☐ a Se reduce en la secuencia creciente, pero no en la decreciente.

☐ b Se reduce en la secuencia decreciente, pero no en la creciente.

☐ c Se reduce en ambas secuencias, tanto la creciente como la decreciente.

☐ d No se reduce en ninguna de estas secuencias.

Respuesta correcta: ☐ d. La TLB reduce el tiempo necesario para realizar las traducciones de dirección lógica a dirección física si ya se han realizado antes, pero no cambia dichas traducciones. Por lo tanto, toda la secuencia se calca con y sin TLB, y el número de fallos va a ser el mismo en ambos casos. Ahora bien, gracias a la TLB las correspondientes traducciones podrán acelerarse y la latencia efectiva que perciba el usuario en el acceso a memoria será algo inferior.

14. Sobre el sistema de memoria inicial recortamos a la mitad la memoria física disponible, es decir, en lugar de 64 bytes se dispone de sólo 32 bytes (4 páginas de 8 palabras cada una). En este caso, para la secuencia de direcciones lógicas {A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24}, la memoria física contendrá los valores correspondientes a las direcciones

☐ a A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24, y el resto de posiciones quedan vacías.

☐ b La segunda mitad de los procesos B y C (esto es, B16 a B31 y C16 a C31), y no queda ninguna posición vacía.

☐ c C0 a C31 (proceso C completo), y no queda ninguna posición vacía.

☐ d Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☐ c. Cada acceso a memoria virtual mete en memoria física la dirección solicitada y las 7 que le acompañan en su misma página. Así, comienzan entrando en memoria física las 4 páginas del proceso A, luego entrarán las 4 páginas del proceso B, que reemplazarán a las del proceso A, y finalmente entrarán las 4 páginas del proceso C, que reemplazarán a las del proceso B.

15. Consideremos que además de recortar a la mitad la memoria física disponible, también reducimos a la mitad el tamaño de página (es decir, disponemos de 32 bytes de memoria física descompuesta en 8 páginas de 4 palabras cada una). En este caso, para la secuencia de direcciones lógicas {A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24}, la memoria física contendrá los valores correspondientes a las direcciones

- ☐ a La segunda mitad de los procesos B y C (esto es, B16 a B31 y C16 a C31), y no queda ninguna posición vacía.
- ☐ b C0 a C31 (proceso C completo), y no queda ninguna posición vacía.
- ☐ c B0 a B3, B8 a B11, B16 a B19, B24 a B27, C0 a C3, C8 a C11, C16 a C19 y C24 a C27.
- ☐ d Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☒ c. Cada acceso a memoria virtual mete en memoria física la dirección solicitada y las 3 que le acompañan en su misma página. Así, comienzan entrando en memoria física las 4 páginas del proceso A, luego entrarán las 4 páginas del proceso B, momento en que se termina de llenar la memoria física. Finalmente entrarán las 4 páginas del proceso C, que reemplazarán a las del proceso A por ser las primeras en entrar.

16. Consideremos que sobre el sistema de memoria inicial reducimos a la mitad el tamaño de página, pero manteniendo la memoria física disponible en 64 bytes (es decir, disponemos de 16 páginas de 4 palabras cada una). En este caso, para la secuencia de direcciones lógicas {A0, A8, A16, A24, B0, B8, B16, B24, C0, C8, C16, C24}, la memoria física contendrá los valores correspondientes a las direcciones

- ☐ a La segunda mitad de los procesos B y C (esto es, B16 a B31 y C16 a C31), y no queda ninguna posición vacía.
- ☐ b C0 a C31 (proceso C completo), y no queda ninguna posición vacía.
- ☐ c B0 a B3, B8 a B11, B16 a B19, B24 a B27, C0 a C3, C8 a C11, C16 a C19 y C24 a C27.
- ☐ d Hay espacio en memoria física para todas las páginas referenciadas. Si la memoria estuviera inicialmente vacía, sobrarían 16 bytes y no sería necesario reemplazar ninguna página.

Respuesta correcta: ☐ d. Cada acceso a memoria virtual mete en memoria física la dirección solicitada y las 3 que le acompañan en su misma página. Como hay 12 peticiones a memoria y cada una está en una página diferente, en total se introducen en memoria física 48 bytes, por lo que quedan libres otros 16 bytes. Los 48 bytes de memoria física estarían compuestos por A0-A3, A8-A11, A16-A19, A24-A27, B0-B3, B8-B11, B16-B19, B24-B27, C0-C3, C8-C11, C16-C19 y C24-C27.

Sea un sistema de memoria con direcciones virtuales de 16 bits montado sobre una memoria física de 4 páginas de 8 palabras de un byte, con algoritmo LRU para el reemplazo de páginas.

17. ¿Cuánto valen las longitudes de los campos **p** para el direccionamiento de la página lógica, **f** para el direccionamiento de la página física y **d** para el desplazamiento de la dirección dentro de la página?

- ☐ a $p=13$, $f=2$, $d=3$.
- ☐ b $p=13$, $f=5$, $d=1$.
- ☐ c $p=16$, $f=2$, $d=3$.
- ☐ d $p=16$, $f=5$, $d=1$.

Respuesta correcta: ☒ a.

- Comenzamos calculando la extensión del campo **d**, que direcciona la palabra dentro de la página, y por lo tanto, su longitud es el logaritmo binario del número de palabras por página. Puesto que hay 8 palabras por página, $8 = 2^3$, resultando $d=3$.

- La dirección virtual es de 16 bits y se compone de los campos **p** y **d** concatenados, por lo que $p+d=16$, resultando $p=13$.
- Finalmente, la longitud **f** es el logaritmo binario del número de páginas de memoria física, esto es, $4 = 2^2$ para darnos $f=2$.

El segmento de código de un programa alojado en la memoria anterior es el siguiente bucle en lenguaje C:

```
int main()
{
    int i, x[100];
    for (i=0; i<100; i++)
        x[i] = i;
}
```

El compilador aloja el contador **i** en un registro interno de la CPU y utiliza 2 bytes de memoria por cada entero del vector **x[]**, resultando el siguiente patrón de acceso a las direcciones pares de memoria dentro del segmento de datos del programa (**A** = *Address* - puntero o dirección de memoria): **A0, A2, A4, A6, ..., A196, A198**.

Dado que el programa apenas tiene un par de instrucciones, nos olvidaremos del segmento de código y simplificaremos suponiendo que toda la memoria física se dedica a alojar el segmento de datos del programa, esto es, el vector **x[]**, y que ninguna de estas páginas ha sido solicitada previamente, por lo que no existe la posibilidad de encontrarla en memoria física cuando comienza a ejecutarse el programa. Bajo estas condiciones, se pide:

18. Calcular el número de faltas de página que se producen durante el acceso a los datos del vector **x[]** cuando se ejecuta el programa.

- ☐ a) 13.
☐ b) 25.
☐ c) 50.
☐ d) 100.

Respuesta correcta: ☒ b). En una página de memoria caben 4 datos de tipo **int**. Los 100 elementos del vector **x[]** caben en 25 páginas, por lo que al recorrerlas todas de forma secuencial sin reutilizar ninguna desde el bucle del programa se producen 25 faltas de página para introducirlas en memoria física o DRAM. La siguiente tabla refleja la página en la que se encuentra cada elemento del vector **x[]** que es accedido desde el bucle:

El elemento	x[0]	x[1]	x[2]	x[3]	x[4]	...	x[99]
está en la página	0	0	0	0	1	...	24
que contiene los datos	x[0..3]	x[0..3]	x[0..3]	x[0..3]	x[4..7]	...	x[96..99]
¿Falta de página?	Sí	No	No	No	Sí	...	No

19. ¿Cuántas faltas de página se producirían si cambiamos el tipo de datos del vector **x[]** de **int** a **float**? (cada **float** o número real en simple precisión ocupa 4 bytes en memoria, frente a los 2 bytes que ocupa cada dato de tipo **int**)

- ☐ a 13.
- ☐ b 25.
- ☐ c 50.
- ☐ d 100.

Respuesta correcta: ☒ c. En una página de memoria caben 2 datos de tipo `float`. Los 100 elementos del vector `x[]` caben en 50 páginas y no se reutiliza ninguna, por lo que se producen 50 faltas de página para introducirlas en memoria cuando son recorridas secuencialmente por el bucle. La siguiente tabla refleja la página en la que se encuentra cada elemento del vector `x[]` que es accedido desde el bucle:

El elemento	<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>x[3]</code>	<code>x[4]</code>	...	<code>x[99]</code>
está en la página	0	0	1	1	2	...	49
que contiene los datos	<code>x[0..1]</code>	<code>x[0..1]</code>	<code>x[2..3]</code>	<code>x[2..3]</code>	<code>x[4..5]</code>	...	<code>x[98..99]</code>
¿Falta de página?	Sí	No	Sí	No	Sí	...	No

20. ¿Cuántas faltas de página se producirían si cambiamos el tipo de datos del vector `x[]` de `float` a `double`, donde cada `double` o número real en doble precisión ocupa 8 bytes en memoria?

- ☐ a 13.
- ☐ b 25.
- ☐ c 50.
- ☐ d 100.

Respuesta correcta: ☐ d. En cada página de memoria sólo cabe un dato de tipo `double`. Por lo tanto, los 100 elementos del vector `x[]` se alojan en 100 páginas de memoria, y dado que ninguna de ellas es reutilizada desde el bucle del programa, se producen 100 faltas de página para introducirlas todas en memoria y poder acceder a los datos que contiene el vector íntegro. De nuevo con la ayuda de una tabla lo veremos más claro:

El elemento	<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>x[3]</code>	<code>x[4]</code>	...	<code>x[99]</code>
está en la página	0	1	2	3	4	...	99
que contiene el dato	<code>x[0]</code>	<code>x[1]</code>	<code>x[2]</code>	<code>x[3]</code>	<code>x[4]</code>	...	<code>x[99]</code>
¿Falta de página?	Sí	Sí	Sí	Sí	Sí	...	Sí

21. ¿Cuántas faltas de página se producirían si cambiamos la sentencia `x[i]=i;` del programa anterior por `x[i]=2*i;`?

- ☐ a La mitad.
- ☐ b Las mismas.
- ☐ c El doble.
- ☐ d Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☐ b. Las faltas de página son sensibles a las direcciones que se solicitan desde el programa, no a los datos que se leen o escriben en estas posiciones.

Si cambiamos el segmento de código anterior por el siguiente:

```

int main()
{
    int i, x[100];
    for (i=0; i<50; i++) // Primero recorremos los elementos pares del vector ...
        x[2*i] = i;
    for (i=0; i<50; i++) // ... y luego recorremos los elementos impares
        x[(2*i)+1] = i;
}

```

22. Calcular el número de faltas de página que se producen durante el acceso a los datos del vector `x[]` cuando se ejecuta el programa anterior.

- ☐ a 13.
- ☐ b 25.
- ☐ c 50.
- ☐ d 100.

Respuesta correcta: ☒ c. Al recorrer los elementos pares del vector referenciamos las mismas 25 páginas que al recorrer todos sus índices, puesto que en todas las páginas hay dos índices de `x[]` pares y otros dos impares, y el primer acceso a cualquiera de ellos introduce en memoria la página completa. Al recorrer los elementos impares del vector sucede algo similar, referenciamos 25 páginas, y dado que comenzamos por las primeras y las que quedan en memoria tras recorrer los elementos pares son las últimas, el segundo bucle no reutiliza ninguna página de las que ha usado el primero. Por lo tanto, este segundo bucle también produce 25 faltas de página, y en total, se producen $25 + 25 = 50$ faltas de página para ejecutar el programa completo. Nótese que la respuesta es la misma al margen de que se elija el algoritmo de reemplazo FIFO, LRU o cualquiera de sus aproximaciones que traten de explotar la localidad espacial y temporal en las referencias a memoria. La siguiente tabla refleja la página en la que se encuentra cada elemento del vector `x[]` que es accedido desde el bucle que recorre los elementos pares:

El elemento	<code>x[0]</code>	<code>x[2]</code>	<code>x[4]</code>	<code>x[6]</code>	<code>x[8]</code>	...	<code>x[98]</code>
está en la página	0	0	1	1	2	...	24
que contiene los datos	<code>x[0..3]</code>	<code>x[0..3]</code>	<code>x[4..7]</code>	<code>x[4..7]</code>	<code>x[8..11]</code>	...	<code>x[96..99]</code>
¿Falta de página?	Sí	No	Sí	No	Sí	...	No

Y esta otra tabla refleja el posterior recorrido de los elementos impares desde el segundo bucle:

El elemento	<code>x[1]</code>	<code>x[3]</code>	<code>x[5]</code>	<code>x[7]</code>	<code>x[9]</code>	...	<code>x[99]</code>
está en la página	0	0	1	1	2	...	24
que contiene los datos	<code>x[0..3]</code>	<code>x[0..3]</code>	<code>x[4..7]</code>	<code>x[4..7]</code>	<code>x[8..11]</code>	...	<code>x[96..99]</code>
¿Falta de página?	Sí	No	Sí	No	Sí	...	No

23. ¿Cuántas faltas de página se producirían al ejecutar el programa anterior si cambiamos el tipo de datos del vector `x[]` de `int` a `float`, donde cada `float` o número real en simple precisión ocupa 4 bytes en memoria?

- ☐ a 13.
- ☐ b 25.
- ☐ c 50.

☒ d) 100.

Respuesta correcta: ☒ d). En una página de memoria caben 2 datos de tipo `float`, un índice par de `x[]` y un índice impar de `x[]` (el que le sigue numéricamente). Los 100 elementos del vector `x[]` caben en 50 páginas y no se reutiliza ninguna, por lo que se producen 50 faltas de página para introducirlas todas en memoria. El recorrido del primer bucle por los índices pares necesita estas 50 páginas, y el del segundo bucle por los índices impares necesita las mismas 50 páginas sin poder reutilizar ninguna (el primer bucle termina con las 4 últimas páginas de `x[]` en memoria física, y el segundo bucle comienza necesitando las 4 primeras, que provocan nuevas faltas de página reemplazando a las 4 últimas). A partir de ahí, el resto de iteraciones del segundo bucle van solicitando nuevas páginas, y todas ellas generan nuevas faltas de página. Por lo tanto, en total se producen $50 + 50 = 100$ faltas de página para ejecutar el programa. La siguiente tabla refleja la página en la que se encuentra cada elemento del vector `x[]` que es accedido desde el bucle que recorre los elementos pares:

El elemento	<code>x[0]</code>	<code>x[2]</code>	<code>x[4]</code>	<code>x[6]</code>	<code>x[8]</code>	...	<code>x[98]</code>
está en la página	0	1	2	3	4	...	49
que contiene los datos	<code>x[0..1]</code>	<code>x[2..3]</code>	<code>x[4..5]</code>	<code>x[6..7]</code>	<code>x[8..9]</code>	...	<code>x[98..99]</code>
¿Falta de página?	Sí	Sí	Sí	Sí	Sí	...	Sí

Y esta otra tabla refleja el posterior recorrido de los elementos impares desde el segundo bucle:

El elemento	<code>x[1]</code>	<code>x[3]</code>	<code>x[5]</code>	<code>x[7]</code>	<code>x[9]</code>	...	<code>x[99]</code>
está en la página	0	1	2	3	4	...	49
que contiene los datos	<code>x[0..1]</code>	<code>x[2..3]</code>	<code>x[4..5]</code>	<code>x[6..7]</code>	<code>x[8..9]</code>	...	<code>x[98..99]</code>
¿Falta de página?	Sí	Sí	Sí	Sí	Sí	...	Sí

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, j, x[100];
    for (j=0; j<10; j++)
        for (i=0; i<10; i++)
            x[10*i] = i;
}
```

24. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector `x[]` mientras se ejecuta el programa?

☐ a) 13.

☐ b) 25.

☐ c) 50.

☒ d) 100.

Respuesta correcta: ☒ d). Puesto que en una página caben 4 datos de tipo `int` y los índices de `x[]` que se solicitan desde el programa están separados 10 unidades, cada acceso que se realiza a ese vector está en una página diferente. Así, cada vez que se recorre el bucle interno `i` se producen 10 faltas de página. Y por cada nueva iteración del bucle externo `j` se repite esta misma secuencia,

sin poderse reutilizar ninguna página (una pasada del bucle interno finaliza con 4 páginas en memoria física que corresponden a la segunda mitad del vector $x[]$, y la siguiente pasada comienza solicitando 4 páginas de la primera mitad del vector). Por lo tanto, el número total de faltas de página que se producen al ejecutar los dos bucles del programa es de $10 * 10 = 100$. La siguiente tabla (dividida en dos partes para que quepa en el folio) refleja la página en la que se encuentra cada elemento del vector $x[]$ que es accedido desde el bucle interno i :

El elemento	$x[0]$	$x[10]$	$x[20]$	$x[30]$	$x[40]$
está en la página	0	2	5	7	10
que contiene los datos	$x[0..3]$	$x[8..11]$	$x[20..23]$	$x[28..31]$	$x[40..43]$
¿Falta de página?	Sí	Sí	Sí	Sí	Sí

El elemento	$x[50]$	$x[60]$	$x[70]$	$x[80]$	$x[90]$
está en la página	12	15	17	20	22
que contiene los datos	$x[48..51]$	$x[60..63]$	$x[68..71]$	$x[80..83]$	$x[88..91]$
¿Falta de página?	Sí	Sí	Sí	Sí	Sí

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, j, x[100];
    for (j=0; j<10; j++)
        for (i=0; i<10; i++)
            x[(10*i)+j] = i;
}
```

25. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector $x[]$ mientras se ejecuta el programa?

- ☐ a) 13.
- ☐ b) 25.
- ☐ c) 50.
- ☐ d) 100.

Respuesta correcta: ☒ d). Los índices de $x[]$ que se solicitan desde dos iteraciones consecutivas del bucle interno están separados 10 unidades, por lo que cada acceso que se realiza a ese vector está en una página diferente. Así, cada vez que se recorre el bucle interno se producen 10 faltas de página. Por cada iteración del bucle externo se repite una secuencia similar, sin poderse reutilizar ninguna página (una pasada del bucle interno finaliza con 4 páginas en memoria física que corresponden a la segunda mitad del vector $x[]$, y la siguiente pasada comienza solicitando 4 páginas de la primera mitad del vector). Por lo tanto, el número total de faltas de página que se producen al ejecutar los dos bucles del programa es de $10 * 10 = 100$. La siguiente tabla (dividida en dos partes para que quepa en el folio) refleja la página en la que se encuentra cada elemento del vector $x[]$ que es accedido desde el bucle interno i para la tercera iteración $j = 2$ del bucle externo (el resto

de iteraciones son similares):

El elemento	x[2]	x[12]	x[22]	x[32]	x[42]
está en la página	0	3	5	8	10
que contiene los datos	x[0..3]	x[12..15]	x[20..23]	x[32..35]	x[40..43]
¿Falta de página?	Sí	Sí	Sí	Sí	Sí

El elemento	x[52]	x[62]	x[72]	x[82]	x[92]
está en la página	13	15	18	20	23
que contiene los datos	x[52..55]	x[60..63]	x[72..75]	x[80..83]	x[92..95]
¿Falta de página?	Sí	Sí	Sí	Sí	Sí

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, j, x[100];
    for (j=0; j<10; j++)
    {
        x[j] = 0;
        for (i=0; i<10; i++)
            x[(10*i)] = i;
    }
}
```

26. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector x[] mientras se ejecuta el programa?

- ☐ a) 50.
- ☐ b) 104.
- ☐ c) 110.
- ☐ d) 200.

Respuesta correcta: ☒ b). Los accesos del patrón `x[j]=0`; introducen 4 accesos a la primera página del vector en las 4 primeras iteraciones del bucle externo, 4 accesos a la segunda página del vector en las 4 iteraciones siguientes, y finalmente, 2 accesos a la tercera página del vector en las dos últimas iteraciones. Estas faltas de página se amortizan luego en los accesos del bucle interno, dado que este bucle empieza siempre solicitando las primeras páginas del vector. Por lo tanto, se produce el mismo número de faltas de página que en la cuestión anterior, ya que cada falta generada por el acceso al índice `j` trae una página a memoria que ya no genera una nueva falta cuando sea accedida seguidamente al comenzar el bucle `i` con el índice `10*i`. No obstante, la excepción a este comportamiento general se produce en las iteraciones 4, 5, 6 y 7 del bucle externo, en las que los accesos del patrón `x[j]=0`; acceden a la segunda página del vector, que luego no es usada por el bucle interno (que accede a la primera página con el índice `x[10*0]=0`; en su primera iteración, y a la tercera página con el índice `x[10*1]=0`; en su segunda iteración, saltándose así la segunda página del vector). Esto produce 4 faltas de página adicionales a las 100 ya comentadas en la solución anterior.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, x[100];
    for (i=0; i<100; i++)
        x[i] = i;
    for (i=99; i>=0; i--)
        x[i] = i;
}
```

27. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector `x[]` mientras se ejecuta el programa?

- ☐ a 25.
- ☒ b 46.
- ☐ c 50.
- ☐ d 100.

Respuesta correcta: ☒ b. En una página de memoria caben 4 datos de tipo `int`. Los 100 elementos del vector `x[]` caben en 25 páginas y no se reutiliza ninguna, por lo que la ejecución del primer bucle produce 25 faltas de página. El segundo bucle también necesita 25 páginas, pero dado que comienza accediendo a las 4 últimas, que son las que ha dejado en memoria la ejecución del primer bucle, sólo produce 21 faltas de página. Por lo tanto, se producen en total $25+21 = 46$ faltas de página. Una vez más, el número de faltas de página es insensible al algoritmo de reemplazo de páginas, que puede ser FIFO, LRU o cualquier otro similar que trate de explotar la localidad espacial y temporal en las referencias a memoria que efectúa el programa.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, j, x[100];
    for (j=0; j<10; j++)
    {
        for (i=0; i<100; i++)
            x[i] = i;
        for (i=99; i>=0; i--)
            x[i] = j;
    }
}
```

28. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector `x[]` mientras se ejecuta el programa?

- ☐ a 110.
- ☐ b 424.

☐ c 1000.☐ d 2000.

Respuesta correcta: ☒ b. En una página de memoria caben 4 datos de tipo `int`. Los 100 elementos del vector `x[]` caben en 25 páginas y no se reutiliza ninguna, por lo que la ejecución del primer bucle `i` para la primera iteración del bucle `j` (`j=0`), se producen 25 faltas de página. La ejecución del segundo bucle para la primera iteración del bucle `j` también necesita 25 páginas, pero dado que comienza accediendo a las 4 últimas, que son las que ha dejado en memoria la ejecución del primer bucle, sólo produce 21 faltas de página. A partir de ahí, cada una de las nuevas ejecuciones del primer y segundo bucle `i` aprovechan las primeras 4 páginas a las que acceden, y producen 21 faltas de página en los accesos restantes. Por lo tanto, son 20 bucles `i` los que se ejecutan, generando 25 faltas de página el primero y 21 faltas de página los 19 bucles restantes. En total, tenemos $25 + (21 * 19) = 424$ faltas de página.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, j, x[100];
    for (j=0; j<5; j++)
        for (i=0; i<5; i++)
            x[i+j] = i+j;
}
```

29. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector `x[]` mientras se ejecuta el programa?

☐ a 3.☐ b 10.☐ c 20.☐ d 100.

Respuesta correcta: ☒ a. El programa referencia a 25 datos del vector `x[]`, pero muchos de ellos se repiten, existiendo sólo 9 datos distintos (el primero referenciado es `x[0]` y el último es `x[4+4]`). Estos 9 datos se encuentran dentro de las 3 primeras páginas del vector, por lo que una vez se generen las 3 primeras faltas de página, los restantes accesos a memoria reutilizarán sus datos y no producirán nuevas faltas de página. Si replicamos la secuencia de peticiones a memoria, son los 3 datos enmarcados los únicos que generan una falta de página cuando se emite su dirección de acceso a memoria: `x[0]`, `x[1]`, `x[2]`, `x[3]`, `x[4]`, `x[1]`, `x[2]`, `x[3]`, `x[4]`, `x[5]`, `x[2]`, `x[3]`, `x[4]`, `x[5]`, `x[6]`, `x[3]`, `x[4]`, `x[5]`, `x[6]`, `x[7]`, `x[4]`, `x[5]`, `x[6]`, `x[7]`, `x[8]`

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
```

```
int i, j, x[100];
for (j=0; j<5; j++)
    for (i=0; i<5; i++)
        x[i*j] = i*j;
}
```

30. ¿Cuántas faltas de página se producen ahora durante el acceso a los datos del vector `x[]` mientras se ejecuta el programa?

- ☐ a 5.
- ☐ b 10.
- ☐ c 25.
- ☐ d Ninguna de las respuestas anteriores es correcta.

Respuesta correcta: ☒ a. Todos los datos que referencia el programa están entre los 17 primeros del vector `x[]`, puesto que el menor índice referenciado es `x[0]` (para `i=0` y `j=0`) y el mayor índice referenciado es `x[4*4]=16` (para `i=4` y `j=4`). Dado que caben 4 datos del vector `a[]` en una página, todos los datos referenciados caben en las 4 páginas disponibles en memoria, salvo el último dato que referencia el programa, `x[16]`, que es el único que pertenece a la quinta página. Por lo tanto, este último acceso provoca el único reemplazo de página, y todos los anteriores se resuelven introduciendo las 4 primeras páginas del vector en memoria, que generan otras 4 faltas de página, para un total de 5 durante la ejecución completa del programa.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, x[100], y[100], z[100];
    for (i=0; i<100; i++)
    {
        x[i] = i;
        y[i] = i;
        z[i] = i;
    }
}
```

31. ¿Cuántas faltas de página se producen durante el acceso a los datos de los vectores `x[]`, `y[]`, `z[]` mientras se ejecuta el programa?

- ☐ a 300.
- ☐ b 150.
- ☐ c 75.
- ☐ d 25.

Respuesta correcta: ☒ c. En una página de memoria caben 4 datos de tipo `int`. Los 100 elementos del vector `x[]` caben en 25 páginas, por lo que al recorrerlas todas de forma secuencial sin reutilizar ninguna desde el bucle del programa se producen 25 faltas de página para introducirlas en memoria física o DRAM. El comportamiento es exactamente el mismo para las páginas del vector `y[]` y para las del vector `z[]`, por lo que el acceso a cada vector genera 25 faltas de página y se producen 75 en total.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, x[100], y[100], z[100];
    for (i=0; i<100; i++)
        x[i] = i;
    for (i=0; i<100; i++)
        y[i] = i;
    for (i=0; i<100; i++)
        z[i] = i;
}
```

32. ¿Cuántas faltas de página se producen durante el acceso a los datos de los vectores `x[]`, `y[]`, `z[]` mientras se ejecuta el programa?

- ☒ a 300.
- ☐ b 150.
- ☐ c 75.
- ☐ d 25.

Respuesta correcta: ☒ c. Se produce el mismo número de faltas de página que en el caso anterior, 25 por cada vector accedido (que son las páginas necesarias para almacenar sus 100 elementos). El recorrido que efectúa el programa por las páginas es diferente en este caso, pero sigue barriéndose el mismo área de datos teniendo que introducir todas sus páginas en memoria, por lo que todas ellas generarán una falta de página la primera vez que son solicitadas.

Si cambiamos el segmento de código anterior por el siguiente:

```
int main()
{
    int i, x[100], y[100], z[100];
    for (i=0; i<100; i++)
        x[i] = i;
    for (i=99; i>=0; i--)
        y[i] = i;
    for (i=0; i<100; i++)
        z[i] = i;
}
```

33. ¿Cuántas faltas de página se producen durante el acceso a los datos de los vectores $x[]$, $y[]$, $z[]$ mientras se ejecuta el programa?

☐ a 300.

☐ b 150.

☐ c 75.

☐ d 25.

Respuesta correcta: ☒ c. Igualmente, se produce el mismo número de faltas de página que en el caso anterior, 25 por cada vector accedido (que son las páginas necesarias para almacenar sus 100 elementos). La única diferencia es que las páginas del vector $y[]$ entran en el orden inverso, pero siguen generando las mismas faltas de página porque no hay localidad de acceso.