



-
1. ¿Cuántos procesos se crean en el siguiente código (incluido el proceso inicial)? Dibuja el árbol de procesos etiquetando sus nodos como A, B y C según representen los correspondientes puntos del código.

```
int main() {
    fork(); // A
    fork(); // B
    fork(); // C
    return 0;
}
```

2. ¿Cuántos procesos se crean en el siguiente código (incluido el proceso inicial)? Dibuja el árbol de procesos.

```
int main() {
    pid_t pid;
    int i;
    for (i=0; i<n; i++) {
        pid = fork();
        if (pid > 0)
        {
            pid = wait(NULL);
            exit(0);
        }
        else
            printf("¡Hola papá!\n");
    }
}
```

3. Sabiendo que `sleep(X)` hace esperar a un proceso X segundos, indicar qué palabra escribe en pantalla el siguiente código C. Puedes ayudarte del árbol de procesos.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void imprime(char *C)
{
    printf("%s",C);
    fflush(stdout);
}

int main() {
    pid_t pid, pid2;
    pid = fork();
    if (pid > 0) {
        imprime("A");
        pid2 = fork();
        imprime("C");
        sleep(1);
        if (pid2 > 0) {
            imprime("I");
            wait(NULL);
            imprime("N");
            wait(NULL);
            exit(0);
        }
        sleep(1);
        imprime("O");
    }
    else
    {
        sleep(3);
        imprime("?");
    }
}
```

4. Sabiendo que `sleep(X)` hace esperar a un proceso X segundos, indicar qué palabra escribe en pantalla el siguiente código C. Puedes ayudarte del árbol de procesos.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void imprime(char *C)
{
    printf("%s",C);
    fflush(stdout);
}

int main() {
    pid_t pid, pid2;
    pid = fork();
    if (pid > 0) {
        sleep(1);
        imprime("A");
        wait(NULL);
        fork();
        imprime("R");
        sleep(1);
        pid2 = fork();
        imprime("I");
        if (pid2 > 0) {
            sleep(2);
            imprime("B");
            wait(NULL);
            imprime("!");
            exit(0);
        }
        sleep(4);
        imprime("A");
    }
    else
        imprime("P");
}
```

5. Determina la salida por pantalla del siguiente programa que lanza el proceso *init*, al que presuponemos PID 1. Considera además que el PID del proceso que está ejecutando el programa es 27, y el de su hijo es 30.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pidfork, mypid, myparent;
    pidfork = fork();
    mypid = getpid();
    myparent = getppid();
    if (pidfork > 0) {
        printf("Parent speaking: First PID is %d.\n", pidfork);
        printf("Parent speaking: Second PID is %d.\n", mypid);
        printf("Parent speaking: Third PID is %d.\n", myparent);
        wait(NULL);
    }
    else {
        sleep(1); // Espera un segundo a que acabe el padre
        printf("Child speaking: First PID is %d.\n", pidfork);
        printf("Child speaking: Second PID is %d.\n", mypid);
        printf("Child speaking: Third PID is %d.\n", myparent);
    }
}
```

6. Determina la salida por pantalla del siguiente programa y razona el resultado:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main() {
    pid_t pid;
    pid = fork();
    if (pid > 0) {
        wait(NULL);
        printf("Parent speaking: value is %d.\n", value);
    }
    else {
        value += 15;
        printf("Child speaking: value is %d.\n", value);
    }
}
```

7. Determina la salida por pantalla del siguiente programa y razona el resultado:

```
#include <sys/types.h>
#include <pthread.h>
#include <stdio.h>

int value = 5;

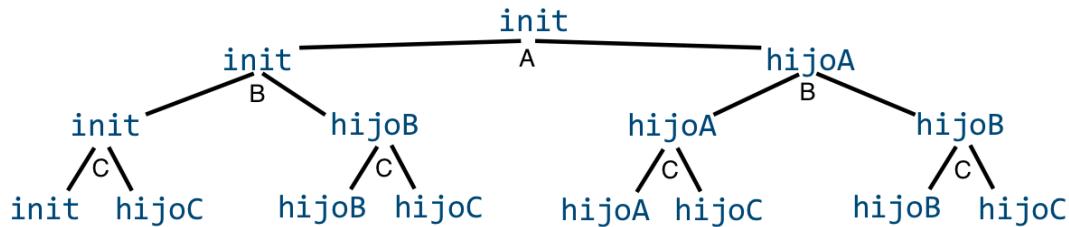
void *runner(void *param) {
    value += 15;
    pthread_exit(0);
}

int main() {
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

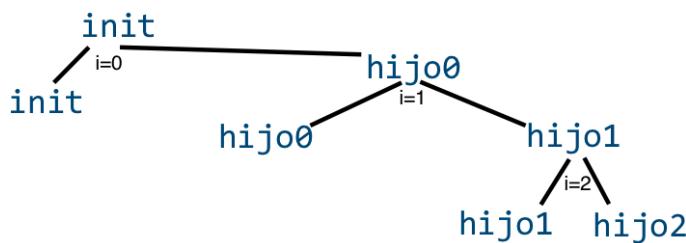
    pid = fork();
    if (pid > 0) {
        wait(NULL);
        printf("Parent speaking: value is %d.\n", value);
    }
    else {
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("Child speaking: value is %d.\n", value);
    }
}
```

SOLUCIONES A LOS EJERCICIOS:

1. Se crean un total de 8 procesos, tal y como muestra el siguiente diagrama:



2. En general, se crea un proceso en cada una de las n iteraciones. Como se indica que se considere también el proceso inicial, la respuesta es $n+1$ procesos, tal y como muestra el siguiente diagrama:

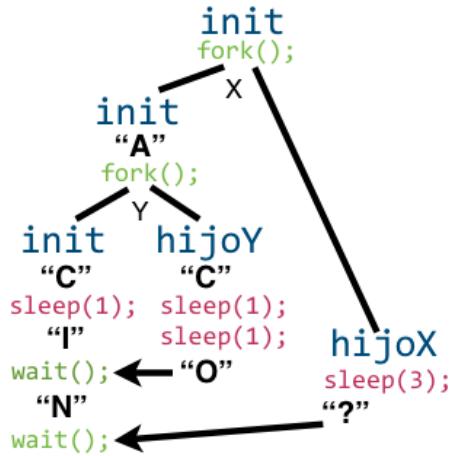


3. Se escribe en la pantalla la palabra ACCION?.

```

pid = fork();           // Creación de un hijo X
if (pid > 0) {
    imprime("A");      // En pantalla: "A"
    pid2 = fork();       // Creación de un hijo Y
    imprime("C");       // En pantalla: "ACC"
    sleep(1);
    if (pid2 > 0) {
        imprime("I");   // En pantalla: "ACCI"
        wait(NULL);      // El padre recoge al hijo Y
        imprime("N");   // En pantalla: "ACCION"
        wait(NULL);      // El padre recoge al hijo X
        exit(0);          // El padre acaba
    }
    sleep(1);            // Por aquí continúa el hijo Y
    imprime("O");       // En pantalla: "ACCIO"
}
else
{
    sleep(3);           // Arranca el hijo X
    imprime("?");      // En pantalla: "ACCION?"
}
    
```

El árbol de procesos es el siguiente:

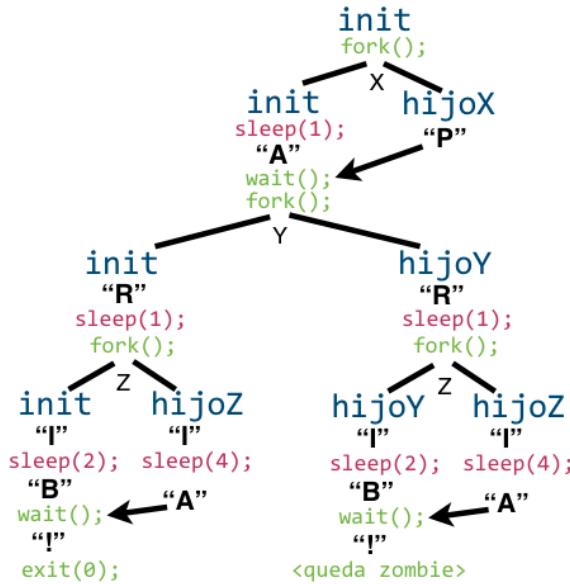


4. Se escribe en la pantalla la palabra PARRIIIBBAA!!.

```

pid = fork();           // Creación de un hijo X
if (pid > 0) {
    sleep(1);
    imprime("A");      // En pantalla: "PA"
    wait(NULL);         // Recogida del hijo X
    fork();              // Creación de un hijo Y
    imprime("R");       // En pantalla: "PARR"
    sleep(1);
    pid2 = fork();       // Creación de dos hijos Z
    imprime("I");       // En pantalla: "PARRIII"
    if (pid2 > 0) {
        sleep(2);
        imprime("B");   // En pantalla: "PARRIIIBB"
        wait(NULL);     // Recogida de los dos hijos Z
        imprime("!");
        exit(0);         // El padre acaba. El hijo Y queda zombie
    }
    sleep(4);            // Por aquí continúan los dos hijos Z
    imprime("A");       // En pantalla: "PARRIIIBBAA"
}
else
    imprime("P");       // En pantalla: "P"
  
```

El árbol de procesos es el siguiente:



5. Se escribe en la pantalla la siguiente secuencia:

Parent speaking: First PID is 30.
 Parent speaking: Second PID is 27.
 Parent speaking: Third PID is 1.
 Child speaking: First PID is 0.
 Child speaking: Second PID is 30.
 Child speaking: Third PID is 27.

Vemos que el padre barre la jerarquía completa, es decir, primero imprime el PID de su hijo, luego el suyo y finalmente el de su padre (que es el proceso `init`). El hijo hace lo mismo, salvo que el primer PID que imprime es 0 (no tiene hijos).

6. Veremos en pantalla los siguientes mensajes:

Child speaking: Value is 20.
 Parent speaking: Value is 5.

Esto es así porque el padre no ve el espacio de direcciones de memoria del hijo. Por lo tanto, el incremento de la variable sólo tiene lugar en la copia local que alberga el hijo.

7. Veremos en pantalla los siguientes mensajes (aunque no necesariamente en ese orden):

Child speaking: Value is 20.
 Parent speaking: Value is 5.

El proceso padre crea un proceso hijo que incrementa el valor de una variable que el padre no puede ver, ya que el hijo actúa en su espacio de direcciones privado (y clonado del padre).

El proceso hijo crea un hilo que incrementa el valor de una variable. Cuando el hilo acaba, el hijo ve ese incremento porque comparte la memoria con dicho hilo.



Ejercicios de Sistemas Operativos. Tema 2: Planificación
Departamento de Arquitectura de Computadores
Ingeniería Informática. Grupo A

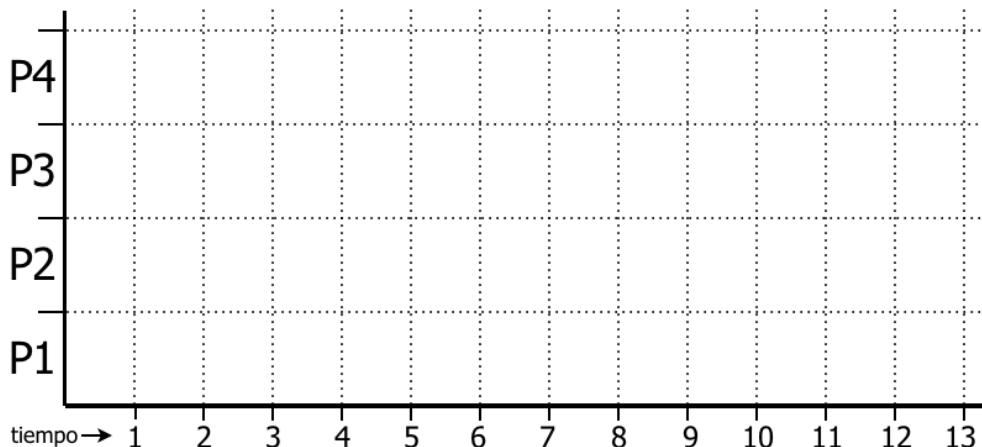


-
1. Sea un sistema de planificación de procesos caracterizado por la siguiente tabla en la que representamos el tiempo de llegada, la duración de las ráfagas de CPU y el período de entrada/salida para cada uno de los procesos, todo ello expresado en milisegundos.

Proceso	Prioridad	Llegada al sistema	Tiempo de CPU	Tiempo de E/S	Tiempo de CPU
P0	4	0	1	1.5	4
P1	3	1	5		
P2	1	2	2		
P3	2	3	1		

Se pide simular la planificación conjunta de estos 4 procesos mediante el diagrama de Gantt adjunto y calcular los tiempos medios de compleción (T_r) y espera (T_e) para ese conjunto de 4 procesos. El ejercicio debe completarse de forma individual para cada uno de los siguientes planificadores:

- a) First Come First Served (FCFS).
- b) Shortest Job First (SJF).
- c) Asignación por prioridad no expropiativa.
- d) Tiempo compartido (*Round Robin*) con un *quantum* de 1 milisegundo.
- e) Shortest Remaining Time First (SRTF).
- f) Prioridad expropiativa.



2. Sea un sistema operativo con planificación de colas de realimentación multinivel en el que se ejecutan tres procesos que llegan al mismo tiempo, estableciéndose el orden de comienzo P1, P2 y P3. P1 requiere 8 ciclos de CPU, P2 10 ciclos y P3 7 ciclos. Tan sólo P2 realiza una operación de entrada/salida, que se completa muy brevemente, gastando un ciclo adicional justo a la conclusión de su tercer ciclo de CPU. La siguiente tabla resume todos estos datos.

	CPU	E/S	CPU
P1	8		
P2	3	1	7
P3	7		

El planificador es expropiativo por prioridad, existiendo tres colas C1, C2 y C3 de mayor a menor prioridad. Los procesos comienzan su ejecución en C1, donde se establece una estrategia *round-robin* (tiempo compartido) con un *quantum* de 2 ciclos. Para C2 se establece *round-robin* con un *quantum* de 3, y para C3 un algoritmo *SRTF* (Shortest Remaining Time First).

Considera las siguientes reglas para el encolado de procesos:

- Si un proceso agota su *quantum*, baja de nivel, y si no lo hace, asciende de nivel.
- Un proceso desalojado por prioridad mantiene ésta, y pasa al final de la cola en la que partió, donde recibirá un *quantum* completo.
- Un proceso que reanuda su ejecución tras un bloqueo también recibe un *quantum* completo.

Se pide lo siguiente:

- a) Simular la planificación de los procesos mediante un diagrama temporal de barras horizontales (Gantt) donde se refleje qué proceso ocupa la CPU en cada momento.
 - b) Si la cola C3 cambiara el algoritmo SRTF por SJF, ¿en qué orden finalizarían los procesos ahora? ¿y si el algoritmo de C3 fuera FCFS (cola FIFO)?
3. Sea un sistema operativo con planificación de colas de realimentación multinivel en el que se ejecutan tres procesos que llegan al mismo tiempo, estableciéndose el orden de comienzo P1, P2 y P3. P1 requiere 8 ciclos de CPU, P2 7 ciclos y P3 10 ciclos. Tan sólo P1 realiza una operación de entrada/salida, que se completa en 4 ciclos justo a la conclusión de su cuarto ciclo de CPU. La siguiente tabla resume la situación.

	CPU	E/S	CPU
P1	4	4	4
P2	7		
P3	10		

Utilizamos un planificador expropiativo con tres colas C1, C2 y C3 de mayor a menor prioridad. Los procesos comienzan su ejecución en C1, donde se establece una estrategia *round-robin* (tiempo compartido) con un *quantum* de 2 ciclos. Para C2 usamos *round-robin* con un *quantum* de 3 ciclos, y para C3 un algoritmo *FCFS* (First Come First Served).

El encolado de procesos se rige por las siguientes normas:

- Si un proceso agota su *quantum* baja de nivel, pero si no lo hace mantiene su nivel.
- Un proceso desalojado por prioridad también mantiene ésta, y pasa al final de la cola en la que partió, donde recibirá un *quantum* completo cuando vuelva a ocupar la CPU.
- Un proceso que reanuda su ejecución tras un bloqueo también recibe un *quantum* completo.

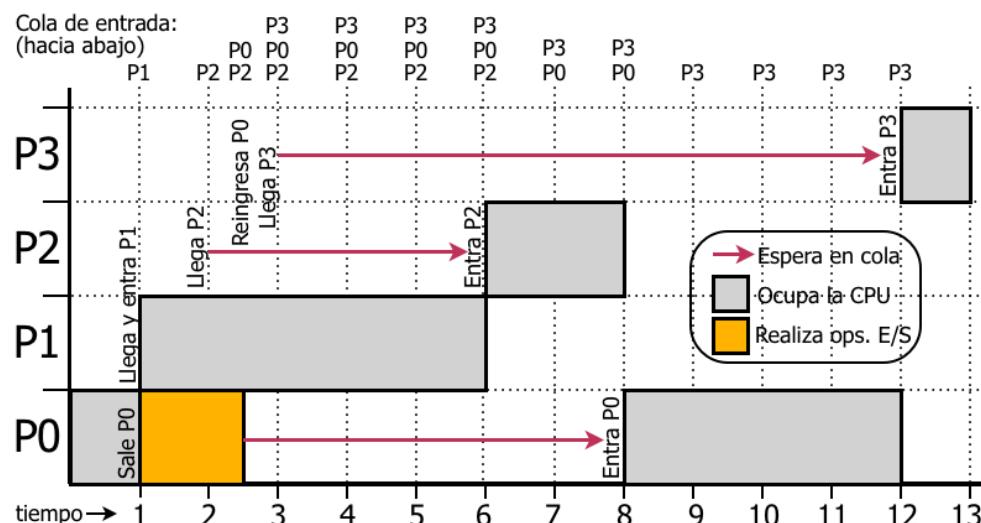
Se pide lo siguiente:

- a) Simular la planificación de los procesos mediante un diagrama temporal de barras horizontales (Gantt) donde se refleje qué proceso ocupa la CPU en cada momento.
- b) Si la cola C3 cambiara el algoritmo FCFS por SJF, ¿en qué orden finalizarían los procesos ahora?

SOLUCIONES A LOS EJERCICIOS:

1. Resolvemos el análisis de cada estrategia de planificación por separado.

a) **Planificación FCFS** (*First Come First Served - el primero que llega es el primero que se atiende*): Según llega cada proceso (o termina una operación de entrada/salida y recupera su estado *Ready*), ingresa en la cola que marca el orden de uso de la CPU. Al ser un algoritmo no expropiativo, el proceso que ocupa la CPU permanece allí hasta que complete su ráfaga de CPU, esto es, hasta que finalice su ejecución o realice una operación de entrada/salida, momento en que es reemplazado por el primero de la cola. El siguiente diagrama ilustra cómo se comparte la CPU siguiendo este algoritmo de planificación, donde la parte superior muestra el estado de la cola que marca el siguiente proceso en ocupar la CPU en cada momento. Lo más significativo que acontece es que cuando P0 finaliza sus 15 milisegundos de entrada/salida ya han llegado P1 y P2, por lo que se reubica detrás de ellos, y poco después llega P3, que ya será el último proceso en ejecutarse.

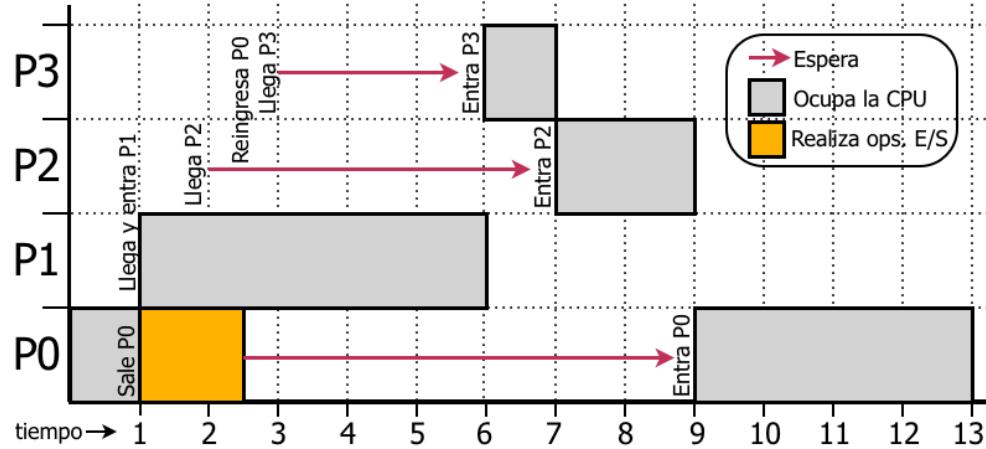


Para calcular los tiempos de compleción (T_r), nos fijamos en la llegada y finalización de cada proceso, restando el primero al segundo. Por otro lado, el tiempo de espera (T_e) se ha marcado para cada proceso con una flecha roja en el diagrama anterior. Todo esto nos lleva a la siguiente tabla, que nos ayuda en el cálculo de los valores medios, $T_r = 82,5$ y $T_e = 46,25$:

Proceso	Llega	Acaba	T_r	T_e
P0	0	12	12	5,5
P1	1	6	5	0
P2	2	8	6	4
P3	3	13	10	9
Valor promedio			8,25	4,62

- b) **Planificación SJF** (*Shortest Job First, primero el trabajo más corto*): En este caso, la planificación de los dos primeros procesos no cambia respecto al caso anterior, puesto que son la única elección del planificador en los instantes 0 y 1. A partir del tercer proceso que llega, momento en el que ya sí debe elegirse, ocupará la CPU aquel proceso que tiene pendiente menos tiempo de CPU. Por lo tanto, el proceso P0, que es el que más tiempo necesita el recurso es relegado al último lugar.

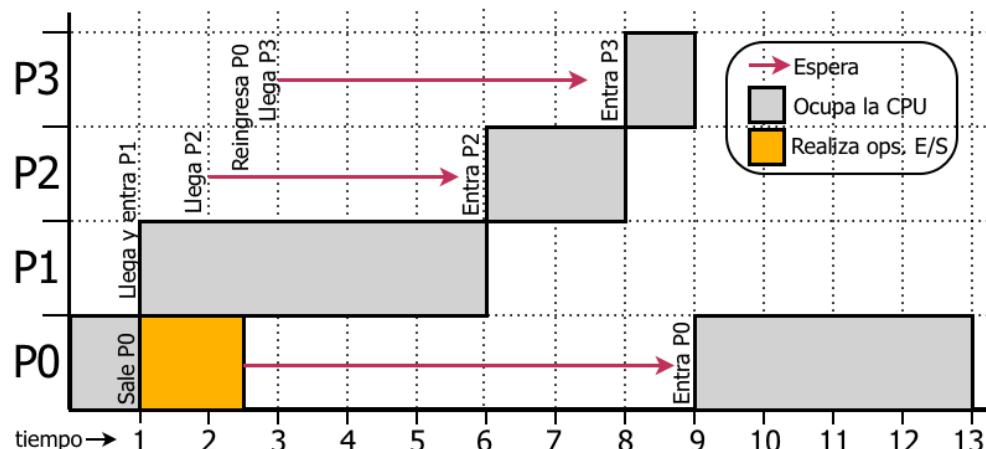
En este caso, no necesitamos mantener la cola de entrada en la parte superior del diagrama de Gantt, pues basta con otorgar la CPU al proceso más corto. La tabla de tiempos se elabora de forma similar al caso anterior con las nuevas cotas temporales de finalización para cada proceso. El resultado es el siguiente:



Proceso	Llega	Acaba	T_r	T_e
P0	0	13	13	6,5
P1	1	6	5	0
P2	2	9	7	5
P3	3	7	4	3
Valor promedio			7,25	3,625

Se aprecia una reducción en el tiempo medio de compleción, T_r , siendo éste el principal objetivo del algoritmo SJF. La espera media, T_e , también se ha reducido, y todo a costa de postergar al proceso P0 (el más extenso), en beneficio de los procesos más ágiles.

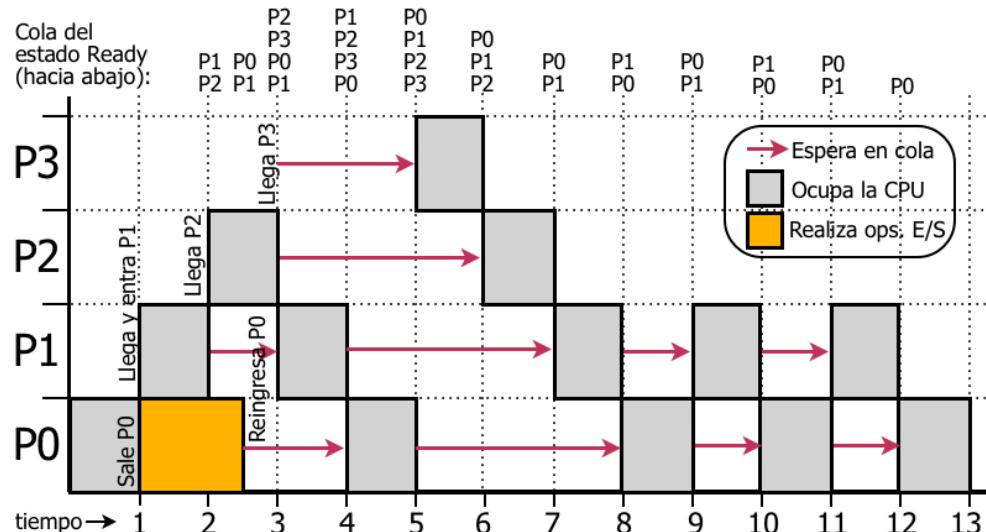
- c) **Planificación por prioridad no expropiativa.** Es quizás el ejercicio más sencillo, puesto que hay que ordenar la ejecución de mayor a menor prioridad, esto es, primero los procesos con números más bajos en la columna de la tabla que nos informa acerca de su prioridad. Eso sí, P0 es de nuevo el primero en tomar la CPU porque, aunque tiene la prioridad más baja, es el único que llega al sistema en el instante inicial. Esa baja prioridad le penalizará cuando finalice sus operaciones de entrada/salida, puesto que para entonces los otros tres procesos ya han entrado al sistema y su ejecución restante queda relegada para la última.



Proceso	Llega	Acaba	T_r	T_e
P0	0	13	13	6,5
P1	1	6	5	0
P2	2	8	6	4
P3	3	9	6	5
Valor promedio			7,5	3,875

- d) **Planificación por tiempo compartido con quantum $q = 1$.** Este método prioriza la justicia en el reparto de recursos frente a la eficiencia, lo que explica que los tiempos promedio sean superiores a los de casos anteriores. Lo más importante para resolver correctamente la asignación de la CPU es que para establecer el turno rotatorio de *quanta*s de tiempo entre los procesos se lleva una cola ligada a los procesos que comparten el estado *Ready*, y cuando un proceso acaba su *quantum* se sitúa al final de dicha cola para dar antes oportunidad a todos los demás.

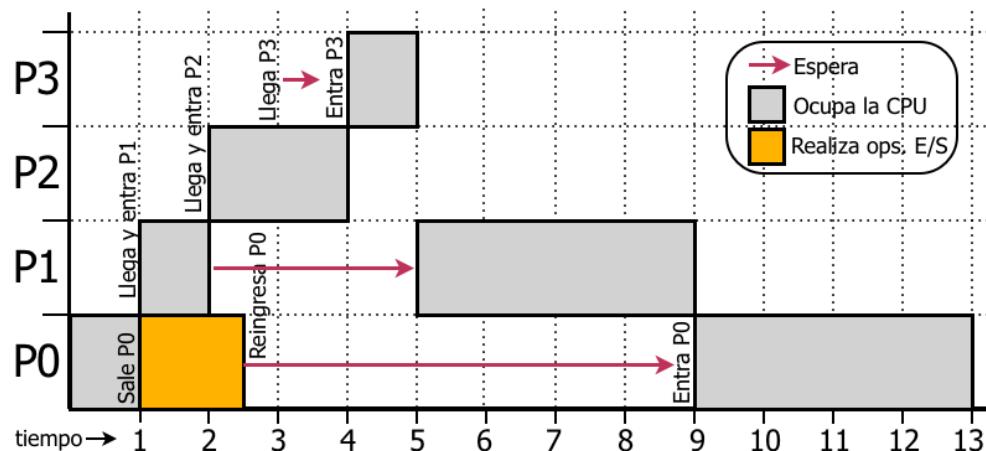
Teniendo todo esto en cuenta, el diagrama de Gantt y la tabla de tiempos son los siguientes:



Proceso	Llega	Acaba	T_r	T_e
P0	0	13	13	6,5
P1	1	12	11	6
P2	2	7	5	3
P3	3	6	3	2
Valor promedio			8	4,375

- e) **Planificación SRTF (Shortest Remaining Time First, primero el trabajo al que le quede menos ráfaga de CPU).** Es la versión expropiativa del método SJF (Shortest Job First), por lo que en cuanto entra un proceso al que le queda menos tiempo de CPU que el que la tiene, le quitará la CPU para quedársela. En nuestra secuencia, esto ocurre cuando llega P2, sacando a P1. P3 es el proceso más corto, pero llega justo cuando a P2 le queda ese mismo tiempo para terminar, por lo que el Sistema Operativo, a igualdad de condiciones, se queda con P2 para no tener que cambiar de contexto. Los grandes perjudicados aquí son los procesos más largos, P0 y P1, que sólo pueden acabar una vez lo han hecho los dos más cortos, P2 y P3.

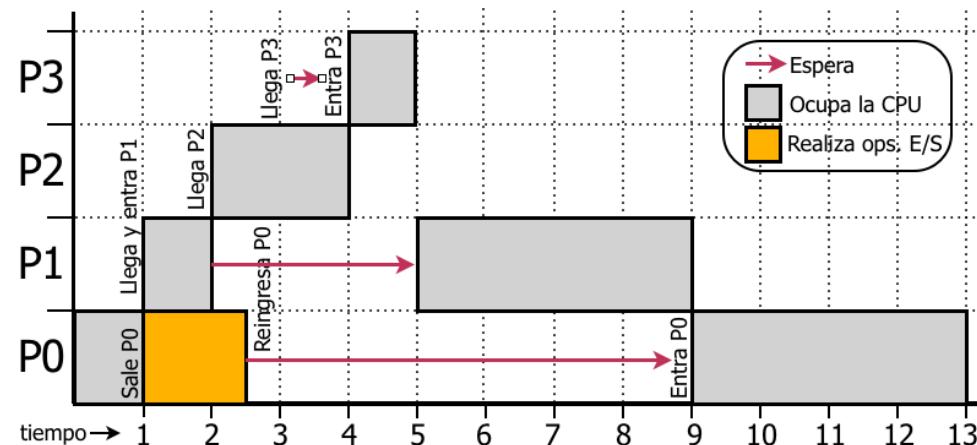
Con todo esto, el diagrama de Gantt y la tabla de tiempos quedan como sigue:



Proceso	Llega	Acaba	T_r	T_e
P0	0	13	13	6,5
P1	1	9	8	3
P2	2	4	2	0
P3	3	5	2	1
Valor promedio			6,25	2,625

f) **Planificación por prioridad expropiativa.** Al igual que la prioridad no expropiativa, la versión expropiativa resulta fácil de simular, ya que basta con fijarse en la columna de la tabla que indica la prioridad. La diferencia entre ambas es que en la primera tomamos la decisión cada vez que acaba el proceso en curso, mientras que aquí lo haremos en cuanto llega un nuevo proceso: Si éste tiene prioridad mayor que el que tiene la CPU (esto es, un número más pequeño), provocará el pertinente relevo en la CPU.

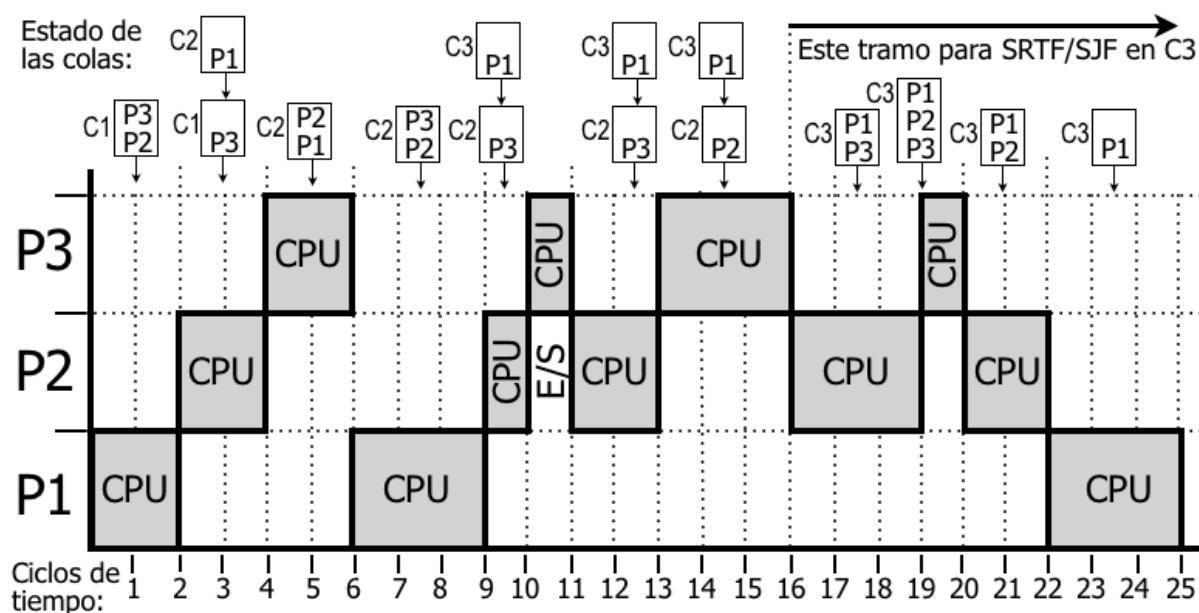
En el caso particular de este ejercicio se da la curiosidad de que las prioridades más altas (números más bajos) se han asignado a los procesos más cortos (P2 y P3), por lo que la asignación de la CPU coincide con el método STRF que acabamos de ver.



Proceso	Llega	Acaba	T_r	T_e
P0	0	13	13	6,5
P1	1	9	8	3
P2	2	4	2	0
P3	3	5	2	1
Valor promedio			6,25	2,625

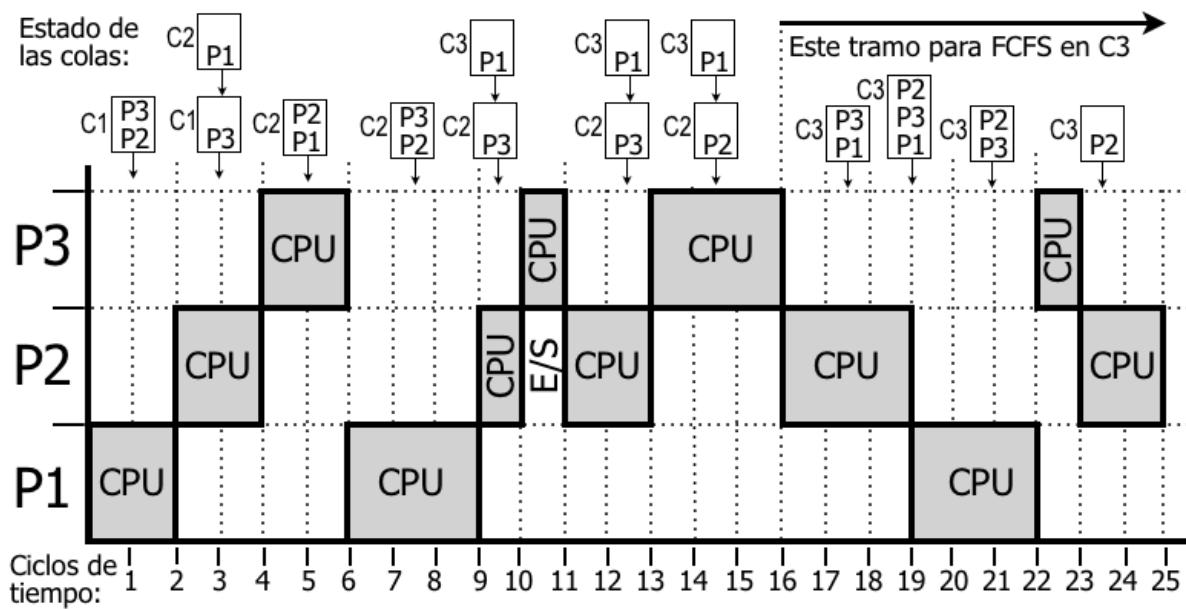
2. En un sistema de planificación de tres colas multinivel, primero se atienden los procesos de la cola 1, y sólo cuando ésta queda vacía comienzan a atenderse los procesos de la cola 2. De forma similar, cuando se agota la cola 2 comienzan a tomarse procesos de la cola 3. De esta manera, podemos considerar que las 3 colas van encadenadas: C1 seguida de C2, y finalmente C3. Así vamos a representarlas en la parte superior del diagrama de Gantt, donde anotaremos todo su movimiento para guiarnos a obtener el siguiente proceso que ocupa la CPU en cada cambio de contexto.

- a) En base a estas premisas y a las reglas que indica el enunciado del ejercicio, la planificación de procesos queda como muestra su diagrama de Gantt a continuación:



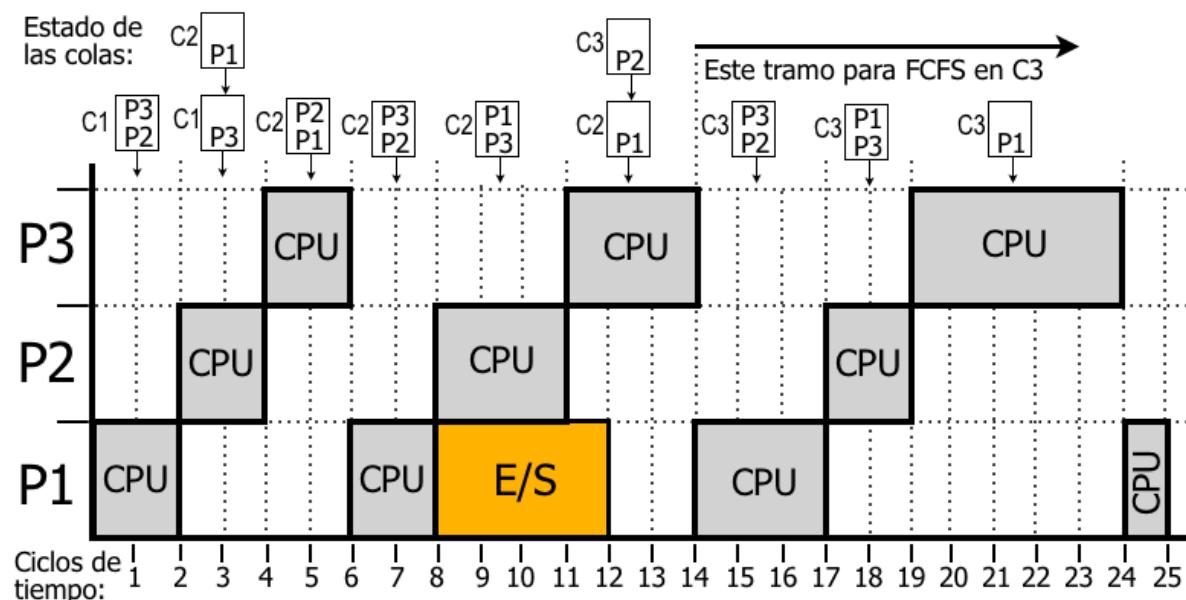
Notar la incorporación del movimiento de los procesos por las 3 colas multinivel en la parte superior. Cada proceso entra a cada cola por arriba y sale por abajo, evolucionando dentro de ella en secuencia vertical.

- b) Respecto a la segunda cuestión que nos plantea cómo finalizan los procesos si la cola C3 cambiara el algoritmo SRTF por SJF o por FCFS, hay que fijarse a partir del ciclo 16 en que el proceso P3 va a parar a la cola C3 donde ya está P1 esperando. Usando SRTF o SJF, P3 se coloca delante de P1, ya que P3 es el proceso al que le quedan menos ciclos de CPU para terminar. En cambio, si usamos FCFS, P3 se colocaría tras P1, puesto que se respeta el orden de llegada (cada nueva entrada se coloca siempre al final). Tomando todas estas consideraciones, la planificación de procesos tomando FCFS en C3 sólo queda alterada a partir de ese ciclo 16, en que se modifica como sigue:



3. Aquí procederemos de forma similar al ejercicio anterior, puesto que las reglas que establecen el movimiento por las colas multinivel son similares. La única diferencia es que en este caso, cuando un proceso no agota su *quantum* de tiempo y es desalojado, se mantiene en la misma cola que estaba en lugar de ascender al siguiente nivel más prioritario.

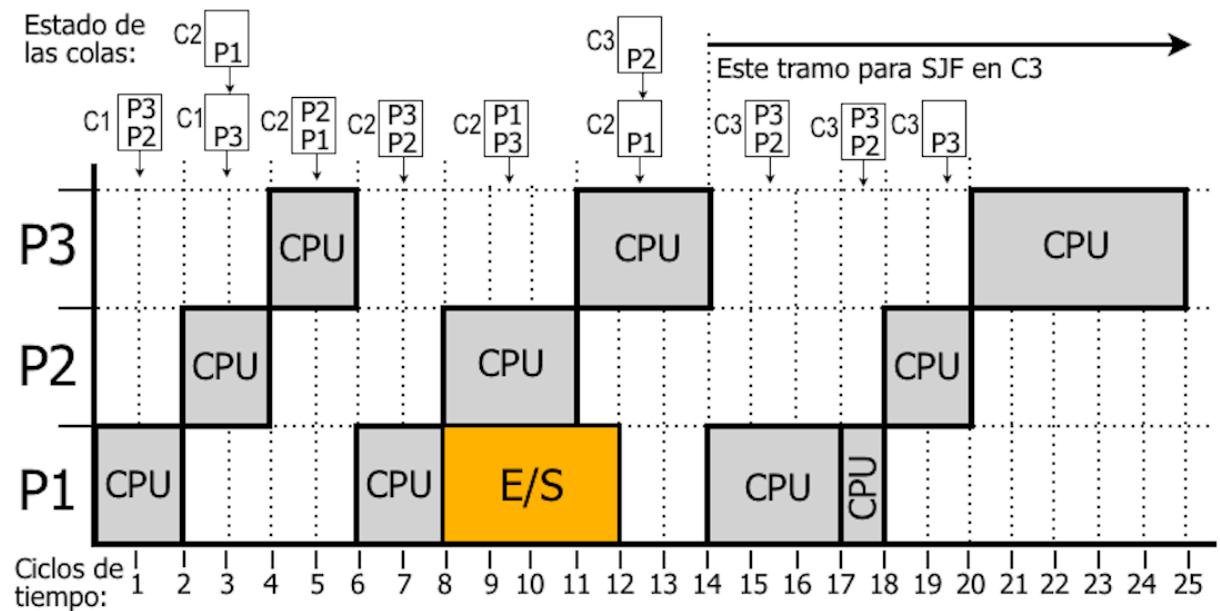
a) El diagrama de Gantt que ilustra la planificación de tareas se muestra a continuación:



En la parte superior del diagrama se detalla todo el movimiento que se produce en las colas que determinan el orden de entrada al sistema.

- b) Respecto a la segunda cuestión que nos plantea cómo finalizan los procesos si la cola C3 cambiara el algoritmo FCFS por SJF, hay que fijarse a partir del ciclo 14, instante en el que empiezan a acumularse procesos en dicha cola. Si la cola se gestiona con FCFS, el orden de finalización sería el orden de entrada en la cola, que ha sido P2, P3 y P1. Pero con SJF se

elige primero el proceso al que queda menos tiempo de CPU por ejecutar, y por lo tanto el orden de finalización sería P1, P2 y P3. El siguiente diagrama muestra los cambios del planificador si se implementara SJF en la cola C3, en el que se aprecian cambios a partir del ciclo 17.





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 quedan 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.

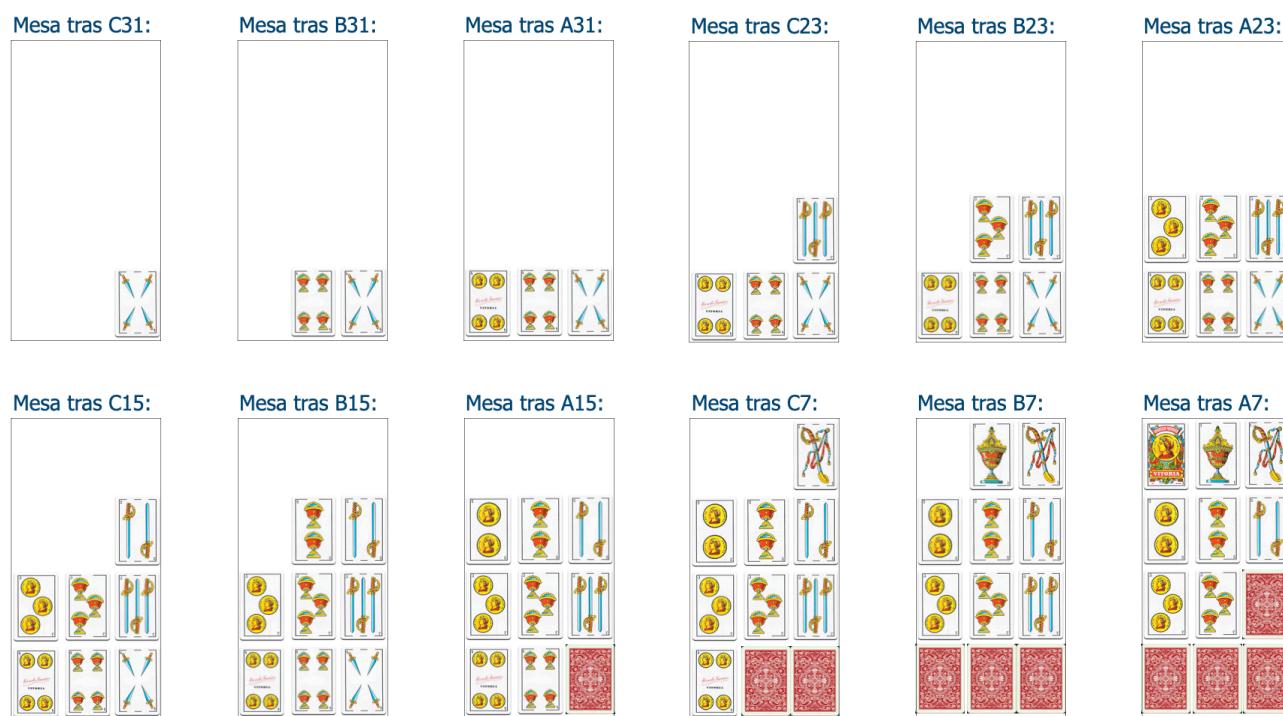
Direcciones de cada proceso agrupadas en sus 4 páginas:	Secuencia de dirs. para la cuestión 1:	Secuencia de dirs. para la cuestión 2:	Secuencia de dirs. para la cuestión 3:	Sugerencia para representar y manipular las páginas en cuestiones sucesivas:
A0 A1 A2 A3 A4 A5 A6 A7 B0 B1 B2 B3 B4 B5 B6 B7 C0 C1 C2 C3 C4 C5 C6 C7	A0 B0 C0 A8 B8 C8 A16 B16 C16 A24 B24 C24	A0 B0 C0 A4 B4 C4 A8 B8 C8 A16 B16 C16 A24 B24 C24	A0 → B0 → C0 A1 A2 A3 A4 A5 A6 A7 B8 B9 B10 B11 B12 B13 B14 B15 C8 C9 C10 C11 C12 C13 C14 C15 B16 B17 B18 B19 B20 B21 B22 B23 C16 C17 C18 C19 C20 C21 C22 C23 B24 B25 B26 B27 B28 B29 B30 B31 C24 C25 C26 C27 C28 C29 C30 C31	 <img alt="Diagram showing a sequence of 8 cards representing memory pages. The first card shows a sequence of 8 boxes labeled A0-A7, B0-B7, and C0-C7.

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):



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 de orden 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, $\{A_0, B_0, C_0, A_1, B_1, C_1, A_2, B_2, C_2, \dots\}$, seguida inmediatamente por la secuencia inversa $\{C_{31}, B_{31}, A_{31}, C_{30}, B_{30}, A_{30}, C_{29}, B_{29}, A_{29}, \dots$ hasta regresar a $A_0\}$. 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.

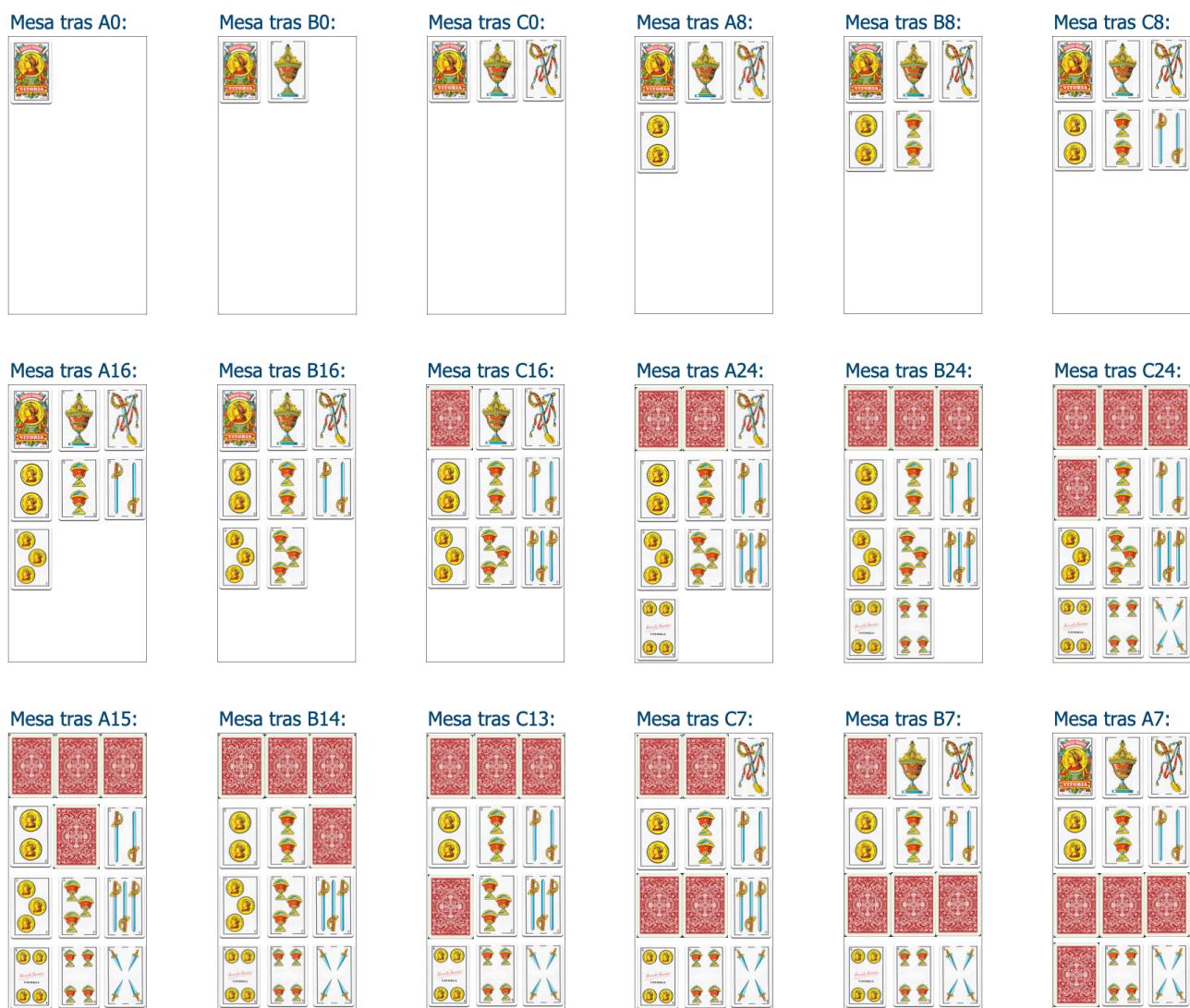
Respuesta correcta: **d**. Sea $P\#X$ la página # del proceso X. Al iniciar la secuencia inversa de peticiones, el contenido de la memoria tiene las 8 páginas siguientes de más a menos antigua (comenzando a numerar en 0): P1B, P1C, P2A, P2B, P2C, P3A, P3B, P3C. A partir de ahí:

- 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



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?

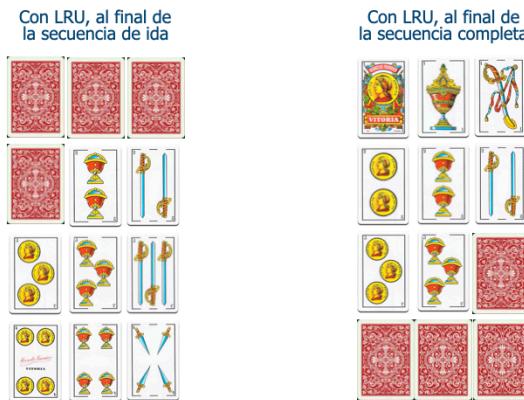
- a) Cambiaría durante la primera mitad de la secuencia (esto es, numeración ascendente), pero no para la segunda mitad (numeración descendente).

- 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.

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.

14. ¿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 dirige 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:

15. 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	<i>x</i> [0]	<i>x</i> [1]	<i>x</i> [2]	<i>x</i> [3]	<i>x</i> [4]	...	<i>x</i> [99]
está en la página	0	0	0	0	1	...	24
que contiene los datos	<i>x</i> [0..3]	<i>x</i> [0..3]	<i>x</i> [0..3]	<i>x</i> [0..3]	<i>x</i> [4..7]	...	<i>x</i> [96..99]
¿Falta de página?	Sí	No	No	No	Sí	...	No

16. ¿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	x[0]	x[1]	x[2]	x[3]	x[4]	...	x[99]
está en la página	0	0	1	1	2	...	49
que contiene los datos	x[0..1]	x[0..1]	x[2..3]	x[2..3]	x[4..5]	...	x[98..99]
¿Falta de página?	Sí	No	Sí	No	Sí	...	No

17. ¿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	x[0]	x[1]	x[2]	x[3]	x[4]	...	x[99]
está en la página	0	1	2	3	4	...	99
que contiene el dato	x[0]	x[1]	x[2]	x[3]	x[4]	...	x[99]
¿Falta de página?	Sí	Sí	Sí	Sí	Sí	...	Sí

18. ¿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;
}
```

19. 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	$x[0]$	$x[2]$	$x[4]$	$x[6]$	$x[8]$...	$x[98]$
está en la página	0	0	1	1	2	...	24
que contiene los datos	$x[0..3]$	$x[0..3]$	$x[4..7]$	$x[4..7]$	$x[8..11]$...	$x[96..99]$
¿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	$x[1]$	$x[3]$	$x[5]$	$x[7]$	$x[9]$...	$x[99]$
está en la página	0	0	1	1	2	...	24
que contiene los datos	$x[0..3]$	$x[0..3]$	$x[4..7]$	$x[4..7]$	$x[8..11]$...	$x[96..99]$
¿Falta de página?	Sí	No	Sí	No	Sí	...	No

20. ¿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	$x[0]$	$x[2]$	$x[4]$	$x[6]$	$x[8]$...	$x[98]$
está en la página	0	1	2	3	4	...	49
que contiene los datos	$x[0..1]$	$x[2..3]$	$x[4..5]$	$x[6..7]$	$x[8..9]$...	$x[98..99]$
¿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	$x[1]$	$x[3]$	$x[5]$	$x[7]$	$x[9]$...	$x[99]$
está en la página	0	1	2	3	4	...	49
que contiene los datos	$x[0..1]$	$x[2..3]$	$x[4..5]$	$x[6..7]$	$x[8..9]$...	$x[98..99]$
¿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;
}
```

21. ¿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;
}
```

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

El elemento	<code>x[52]</code>	<code>x[62]</code>	<code>x[72]</code>	<code>x[82]</code>	<code>x[92]</code>
está en la página	13	15	18	20	23
que contiene los datos	<code>x[52..55]</code>	<code>x[60..63]</code>	<code>x[72..75]</code>	<code>x[80..83]</code>	<code>x[92..95]</code>
¿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;
    }
}
}

```

23. ¿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;
}

```

24. ¿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;
    }
}
```

25. ¿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;
}

```

26. ¿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;
}

```

27. ¿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.



Ejercicios de Sistemas Operativos. Tema 3: Memorias

Ingeniería Informática. Grupo A

Autor: Manuel Ujaldón

Dpto. de Arquitectura de Computadores. UMA



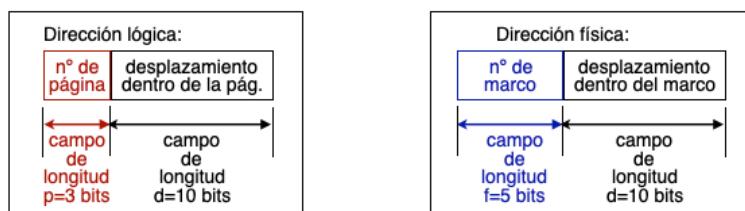
Resumen preliminar. Este capítulo tiene 8 conceptos clave, asociados en 4 parejas afines:

- Cada proceso que accede a memoria emite **direcciones virtuales** (también denominadas *direcciones lógicas*), creando antes su propia memoria o **espacio de direcciones virtuales**.
- El Sistema Operativo se encarga de traducir cada dirección virtual a una **dirección física**, ubicada dentro del **espacio de direcciones físicas**, que es la memoria real de que se dispone.
- Tanto el espacio virtual como el físico se descompone en bloques de tamaño fijo de entre 4 y 16 Kbytes, denominados **páginas** en el primer caso y **marcos (frames)** en el segundo.
- La traducción de cada dirección virtual a su dirección física asociada se realiza a nivel de página, que encontrará su marco asociado en una **tabla de páginas** mantenida por el Sistema Operativo. Para acelerar este proceso, se pueden guardar en una **TLB** (o caché) aquellas parejas de traducciones ya hechas (cada número de página y su número de marco asociado), evitando el acceso a la tabla de páginas cuando son referenciadas de forma frecuente.

-
1. Sea un espacio de direcciones lógico de 8 páginas de 1024 palabras de un byte cada una, mapeado sobre un espacio de direcciones físico de 32 marcos. ¿Cuántos bits de longitud tiene cada dirección lógica y cada dirección física?

La dirección lógica tiene 2^3 páginas de 2^{10} palabras, por lo que debe tener una longitud de $3 + 10 = 13$ bits.

Por otra parte, la dirección física tiene 2^5 marcos de 2^{10} palabras (la página y su marco asociado deben coincidir en tamaño para que éste pueda albergar a aquélla). Por lo tanto, cada dirección física debe tener una longitud de $5 + 10 = 15$ bits. En resumen, tendríamos lo siguiente:



Aclaración: Lo habitual es que la dirección lógica sea más extensa que la dirección física, puesto que cada proceso suele utilizar más memoria virtual de la que se dispone físicamente en el sistema. En el caso que nos ocupa sucede lo contrario, pero aún así, el sistema necesitará memoria virtual en cuanto haya más de 4 procesos, pues en total se superarán las 32 páginas para alojar por completo su espacio de direcciones, y sólo tenemos 32 marcos para ellas en memoria física.

2. Una memoria virtual paginada cuenta con direcciones lógicas de 16 bits y 16 marcos de 1024 palabras de memoria. En un momento dado, su tabla de páginas tiene los siguientes contenidos específicos para sus 64 entradas (en azul):

N	Valor
0	1001
1	0100
2	0000
...	...
62	0101
63	1100

Realizar la traducción a dirección física de las siguientes direcciones de memoria virtual:

- a) 1111111111111111.
- b) 0000000000000000.
- c) 0000100100010001.
- d) 1111100000111111.

El tamaño del marco, 1024 palabras (2^{10}), nos indica que se utilizan 10 bits para direccionar el desplazamiento dentro de la página (y dentro del marco, pues ambos tienen el mismo tamaño). Estos 10 bits son los últimos de la dirección, al ser el campo numéricamente menos significativo. Por lo tanto, los 6 bits superiores de la dirección lógica corresponden a su número de página, y con estos dos campos de 6 y 10 bits debemos realizar la traducción a una dirección física de 14 bits (4 para el marco y 10 para el desplazamiento) de la siguiente forma:

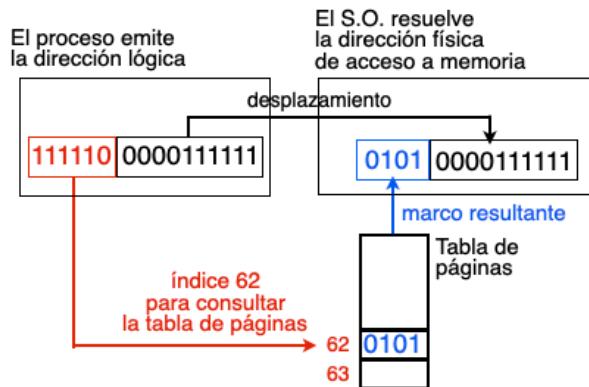
- El número de página (6 bits) debe traducirse al número de marco que contiene dicha página en memoria física utilizando la tabla de páginas anterior, donde la página actúa como índice de acceso a las entradas de la tabla y el marco se corresponde con el valor de dicha entrada.
- El desplazamiento dentro del marco (10 bits) deberá agregarse al número de marco anterior para dar con la ubicación exacta de la dirección de memoria solicitada, que estará compuesta de ambos campos concatenados. Esta concatenación se produce por el peso numérico de cada uno de los campos: Los marcos direccionan en múltiplos de 1K (que por tanto comienzan en una dirección que tiene los 10 últimos bits a cero) y el desplazamiento dentro del marco es el que concreta el valor de estos 10 últimos bits. En decimal quizás lo veamos mejor: si tenemos 500 palabras de memoria física organizada en 50 marcos de 10 palabras cada uno, el marco 27 agrupa desde la dirección 270 a la 279, y así, la dirección 273 estaría en el marco 27, y dentro de él, en la dirección 3 (y concatenando 27 y 3, formamos el número 273).

Lo primero es localizar el marco que corresponde a la página según la tabla, y posteriormente el dato dentro del marco, que se encontrará en la dirección relativa indicada por su desplazamiento. Procedamos de esta forma para cada una de las referencias a memoria, coloreando en rojo el **número de página**, en azul su **número de marco** asociado, y en negro el **desplazamiento** dentro de ambos:

- a) La dirección virtual **1111111111111111** está en la página **63**, desplazamiento **1023** (pasando ambos de binario a decimal). La tabla de páginas indica que la página **63** se encuentra ubicada en el marco **1100** de memoria física (esto es, el **12**), por lo que la dirección física resultante es la concatenación de los campos marco y desplazamiento, esto es, **11001111111111**.

- b) La dirección virtual **0000000000000000** está en la página **0**, desplazamiento **0**. La tabla de páginas indica que la página **0** se encuentra ubicada en el marco **1001** de memoria física (el **9** en numeración decimal), por lo que la dirección física resultante es **10010000000000**. En representación decimal, dirímos que la dirección virtual 0 se ubica en la dirección física 9K, esto es, $9 \times 1024 = 9216$.
- c) La dirección virtual **0000100100010001** está en la página **2**, que según la tabla de páginas se ubica en el marco **0000** de memoria física, por lo que la dirección física resultante es **00000100010001**.
- d) Por último, la dirección virtual **1111100000111111** está en la página **62**, que la tabla de páginas nos dice que se ubica en el marco **0101** de memoria física, resultando la dirección física **01010000111111**.

Para esta última traducción, el esquema visto en clase de forma genérica para abordar las traducciones podría particularizarse de la siguiente forma:



3. Sobre un espacio de direcciones virtuales de 1 Giga-palabras de 1 byte montado sobre una memoria física de 256 Megabytes, calcular:
- El tamaño de página que da lugar a una tabla de páginas de 2^{19} entradas.
 - El número de bits de que se compone cada dirección física.
 - El número de marcos que necesita la memoria física.
-
- La tabla tiene una entrada por cada página de memoria virtual, por lo que debemos tener 2^{19} páginas de X palabras cada una para componer el espacio de direcciones virtuales de 1 Gigapalabra (2^{30} palabras). Despejando X, tenemos un tamaño de página de $X = \frac{2^{30}}{2^{19}} = 2^{11} = 2048$ palabras.
 - La memoria física tiene 256 Mpalabras de 1 byte, y dado que $2^{28} = 256M$, necesitamos 28 bits para direccionar todas esas palabras.
 - El marco y la página tienen el mismo tamaño, que ha resultado ser de 2048 palabras de 1 byte (2^{11} bytes). Para componer el total de la memoria física de 256 Mbytes (2^{28} bytes) son necesarios por tanto 2^{17} marcos de 2^{11} bytes cada uno.

4. Consideremos direcciones virtuales de 64 bits y páginas de 4 Kilobytes sobre una memoria física de 512 Megabytes que direcciona palabras de 32 bits. Calcular el espacio que ocupa en memoria la tabla de páginas considerando que se usan 7 bits de control en cada una de sus entradas.

La tabla de páginas contiene una entrada o fila por cada página de memoria virtual, que se encarga de darnos su traducción al marco que lo aloja en memoria física (y contendrá, adicionalmente, los 7 bits de control).

Cada página de 4 Kbytes alberga 1K palabras de 32 bits (4 bytes). El número de páginas necesario es el cociente entre las 2^{64} direcciones virtuales y las 2^{10} palabras que caben en cada página, es decir, 2^{54} . Ése es el número de filas de la tabla de páginas.

Por otro lado, la memoria física es de 512 Mbytes y sus marcos son de 4 Kbytes, existiendo en total $\frac{512\text{Mbytes}}{4\text{Kbytes}} = \frac{2^{29}}{2^{12}} = 2^{17}$ marcos de página, por lo que son necesarios 17 bits para direccionarlos. Estos 17 bits constituyen la longitud de cada fila de la tabla de páginas (puesto que contiene el número de marco en el que se aloja cada página), completados con los 7 bits de control para llegar a 24 bits (3 bytes) en cada entrada de la tabla de páginas.

La tabla de páginas tendrá por tanto 2^{54} filas de 3 bytes cada una, es decir, 48 Petabytes. La magnitud de este número nos dice que con una memoria virtual tan extensa es imposible utilizar un sistema paginado de un solo nivel, teniendo que recurrir a una tabla de páginas multinivel. El precio a pagar es la lentitud de las traducciones de dirección virtual a física, puesto que ahora serán necesarios múltiples accesos a memoria física para consultar la tabla de páginas, uno para cada nivel añadido.

5. Un sistema de memoria virtual con páginas de 8 Kilobytes, espacio lógico de 1 Terabyte y espacio físico de 32 Gigabytes, utiliza paginación multinivel para implementar su ingente tabla de páginas. Indica la estructura de dicha tabla de páginas (número de entradas que posee y tamaño que tiene cada una de ellas) según organicemos ésta en múltiples niveles. Partiendo de la tabla en un solo nivel, debes ir paginando la misma e incorporando nuevos niveles sucesivamente más externos para acceder a todas las páginas que resulten del nivel anterior, hasta llegar al nivel en el que todas sus entradas quepan en una sola página. Considera la presencia de 10 bits de control acompañando al número de marco en cada entrada de las tablas de páginas.

La tabla de páginas contiene una entrada por cada página de memoria virtual, que nos indica el marco que aloja sus datos en memoria física (junto con los 10 bits de control que maneja el Sistema Operativo).

Una memoria física de 32 Gigabytes se descompone en 4 Mega-marcos de 8 Kilobytes cada uno (el tamaño del marco debe coincidir con el de la página, puesto que alberga los datos de ésta). Estos 4 Mega-marcos pueden direccionarse usando $f = 22$ bits, ya que $2^{22} = 2^2 \times 2^{20} = 4$ Mega. Con este dato ya podemos concretar la **anchura de las tablas de páginas: 32 bits**, que corresponden con los 22 bits para el número de marco que aloja sus datos y los 10 bits de control para dicho marco (bit de presencia, bit de validación, bit *dirty*, etc).

- La memoria virtual de 1 Terabyte tendrá 128 Mega-páginas de 8 Kilobytes cada una. Puesto que la tabla de páginas tiene una entrada para cada página, dispone de 128 Mega-entradas, que ya sabemos que son de 32 bits. En total, el espacio ocupado por **esta tabla será de 128 Mega-entradas × 4 bytes** si la implementamos en un solo nivel, ocupando un espacio de 512 Mbytes.

- Para incorporar el segundo nivel, procedemos a paginar la tabla anterior. En una página de 8 Kbytes caben 2048 entradas de la tabla (de 4 bytes cada una), por lo que el número de páginas necesarias para esta tabla es de $\frac{128 \text{ Megaentradas}}{2048 \text{ entradas/página}} = 65536$ páginas. También podemos llegar a este dato dividiendo el espacio ocupado por la tabla anterior (512 Mbytes), por el tamaño de cada página (8 Kbytes).

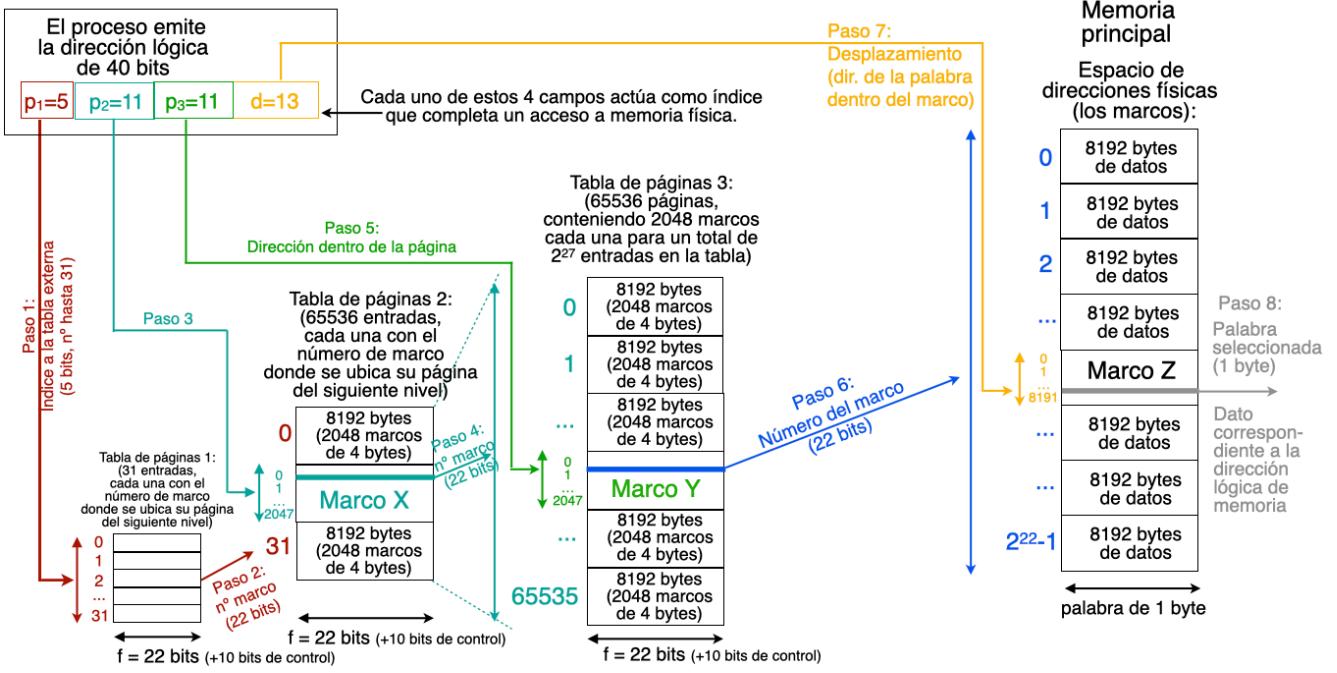
La tabla del segundo nivel debe contener una entrada por cada página de la tabla anterior, en la que se ubicará el marco que la contiene. Esto nos lleva a una **segunda tabla que tiene 65536 entradas de 4 bytes** cada una (de nuevo, los 22 bits del marco y los 10 bits de control), ocupando un espacio total de 256 Kbytes.

- Para incorporar el tercer nivel, paginaremos la tabla anterior de 256 Kbytes. Dividiendo este tamaño por el de la página, 8 Kbytes, obtenemos que son necesarias 32 páginas para alojar la segunda tabla, por lo que **esta tercera tabla ubicada en el nivel más externo tendrá 32 entradas**. Una vez más, cada entrada de esta tabla contendrá 32 bits (el número de marco de 22 bits y los 10 bits de control adicionales). El tamaño de esta tercera tabla es de sólo 128 bytes, por lo que ya cabe dentro de una sola página de memoria, concluyendo aquí la construcción de todas las tablas suplementarias necesarias para acceder al último nivel.

Recopilando todo y reenumerando los niveles para comenzar por el más externo, tenemos:

- El acceso al nivel 1 más externo se realiza con un índice p_1 de 5 bits que accede a una tabla de 32 entradas, donde encontramos el marco que ubica la página de la tabla del siguiente nivel a la que debemos acceder. Esto completa el primer acceso a la memoria física.
- El marco anterior nos sitúa en una página X dentro de la tabla del nivel 2, donde ahora utilizaremos el índice p_2 como desplazamiento desde la base de X para obtener a su vez el marco que ubica la página de la tabla del nivel 3. Así completamos el segundo acceso a memoria física.
- A su vez, el marco anterior nos ubica al comienzo de una página Y dentro de la tabla del nivel 3 (el más interno). Aquí usaremos el índice p_3 como desplazamiento dentro de Y para obtener la dirección física que accede a memoria física por tercera vez.
- Finalmente, el dato de la dirección anterior nos proporciona la dirección base del marco Z, que ya contiene los 8 Kbytes de datos en memoria física. Utilizando el campo d (desplazamiento) como dirección relativa a dicha base, obtendremos la dirección de memoria física donde se encuentra la palabra solicitada por la dirección virtual de partida, completando el cuarto y último acceso a memoria física.

La estructura de cada una de las tablas que componen los índices de acceso a los diferentes niveles se refleja en la siguiente figura conjunta:



6. Un sistema de memoria virtual con palabras de 32 bits habilita 2^{20} marcos de página en una memoria física de 64 Gbytes. La traducción de direcciones virtuales a físicas tiene lugar a través de una tabla de páginas estructurada en dos niveles, cuyos campos de acceso son los siguientes:

Índice al primer nivel	Índice al segundo nivel	Campo desplazamiento dentro de la página
$p_1 = 12$ bits	$p_2 = 14$ bits	$d = ?$

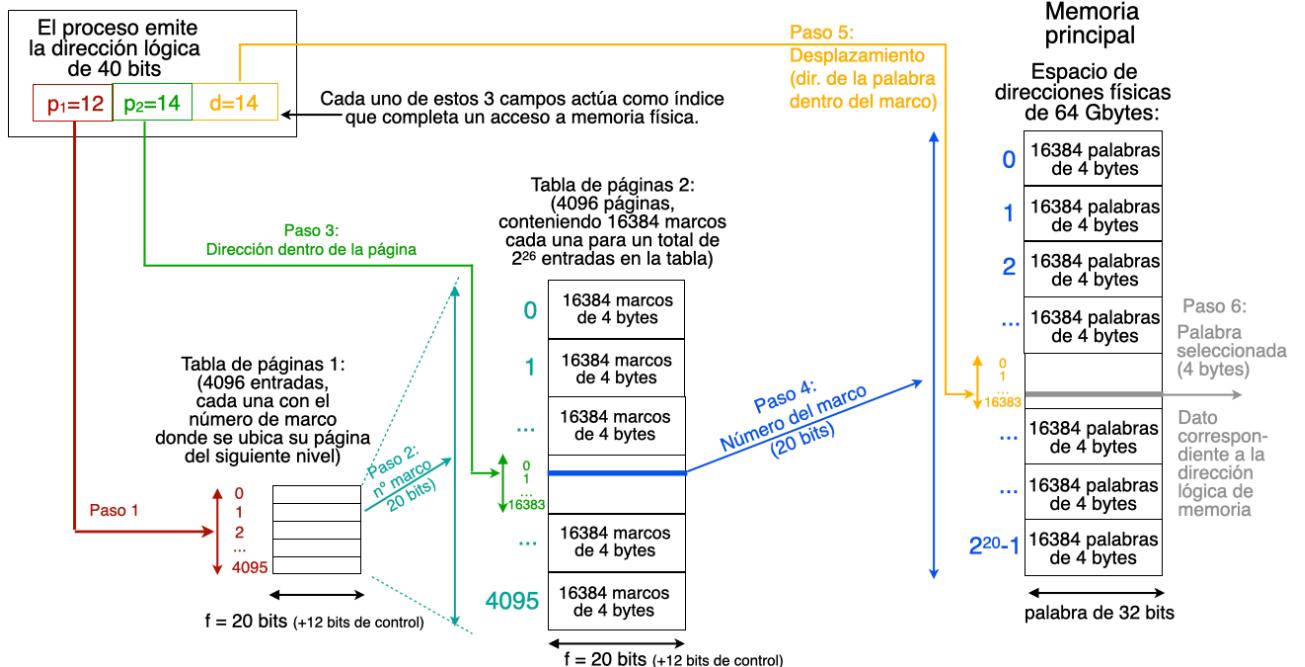
Teniendo en cuenta toda esta información, se pide determinar:

- El tamaño de página.
- La longitud del campo d (desplazamiento) que direcciona la palabra dentro del marco.
- El espacio de direcciones virtuales de cada proceso.
- Si se considera una sola tabla de páginas (un solo nivel), indica cuántas entradas serían necesarias en dicha tabla. Estas entradas tendrán que incorporar, adicionalmente, 12 bits de control cada una.
- Incorpora ahora el segundo nivel de la tabla paginando el anterior e indicando el efecto que tendría en el rendimiento del proceso de traducción respecto al caso anterior.

En esta ocasión, las palabras de memoria no son de un byte como es habitual en un PC actual, sino de 32 bits (4 bytes). Esto supone que la memoria se direcciona en múltiplos de 4 si tomamos el byte como unidad de referencia (es decir, la CPU accede al byte 0, 4, 8, ..., ya que el valor del byte 2, por ejemplo, forma parte de la primera palabra de memoria junto a los valores de los bytes 0, 1 y 3). Más razonable y sencillo resulta, por tanto, adaptar nuestros bytes a palabras y usar ésta como magnitud de referencia, teniendo siempre presente que las cantidades expresadas en bytes se dividen por 4 para saber cuántas palabras de memoria contienen.

- a) El tamaño del marco coincide con el de la página de memoria. Si tenemos 2^{20} marcos en una memoria física de 64 Gigabytes con palabras de 4 bytes, la memoria tendrá 16 Gigapalabras (2^{34} palabras). El cociente entre esta cantidad y el número de marcos, nos dirá cuántas palabras hay en cada marco, y su logaritmo binario será la longitud del campo d utilizado para direccionar la palabra dentro del marco:
- $$\frac{2^{34} \text{ palabras}}{2^{20} \text{ marcos}} = 2^{14} \text{ palabras/marco.}$$
- Por lo tanto, el tamaño de página es de 2^{14} palabras (64 Kbytes) y d=14.
- b) La longitud del campo d (desplazamiento), 14 en este caso, nos dice el número de bits necesarios para direccionar la palabra dentro de la página (o del marco). Este desplazamiento interviene en el último paso del proceso de traducción, una vez se conoce el marco que alberga los datos de la página que contiene la dirección de memoria virtual. El campo d permite localizar la dirección física dentro del marco que contendrá el dato de 32 bits que conforma la palabra de memoria solicitada.
- c) La dirección de memoria virtual tiene un total de $12+14 = 26$ bits para indicar la página (12 bits corresponden al primer nivel y 14 al segundo), y 14 bits adicionales para indicar el desplazamiento dentro de la página. Esto nos da un total de 40 bits de longitud. Por lo tanto, el espacio de memoria virtual se compone de 2^{40} palabras de memoria de 4 bytes, esto es, 4 Terabytes, organizados en 2^{26} páginas de 64 Kbytes.
- d) Para el caso de una sola tabla de páginas, la tabla tiene una entrada para cada página de memoria virtual, esto es, 2^{26} entradas. Cada entrada alberga un número de marco y los 12 bits de control. Como el número de marco ocupa 20 bits, cada entrada de la tabla de páginas ocupará 32 bits (4 bytes). Recopilando todo, el espacio total ocupado por esta tabla es de $2^{26} \text{ entradas} \times 4 \text{ bytes/entrada} = 256 \text{ Mbytes.}$
- e) La tabla anterior de 256 Mbytes se pagina en 4 Kpáginas de 64 Kbytes, y en cada una de esas páginas caben 16 Kmarcos, puesto que cada marco ocupa 32 bits. La segunda tabla de páginas que se incorpora direcciona a estas 4 Kpáginas con los 12 bits más significativos del campo dirección, esto es, p₁. Los 14 bits restantes, correspondientes a p₂, seleccionan la entrada dentro de dicha página, que nos dará la dirección base del número de marco que contiene los datos de la palabra en memoria física. Esta segunda tabla ocupa 4 Kentradas x 32 bits = 16 Kbytes, que ya nos cabe dentro de una sola página, y por lo tanto, no es necesario volver a paginar.

El siguiente diagrama ilustra la ubicación de las tablas que componen los índices de acceso a los marcos de página y su relación con los campos de la dirección virtual emitida desde el proceso:



7. El sistema de memoria de mi PC tiene las siguientes características:

- Sistema Operativo con espacio de direcciones virtuales de 64 Terabytes para cada proceso, direccionable por páginas de 4 Kbytes y palabras de un byte.
- Memoria física (DRAM) de 32 Gbytes.
- Microporcesador con TLB (Translation Look-Aside Buffer) de 8 entradas.

Se pide detallar:

- a) La longitud de los campos p (página virtual), f (marco físico o *frame*), y d (desplazamiento) en las respectivas direcciones de memoria.
- b) La estructura de la TLB, desglosando la información que contienen sus 8 entradas.

Los campos p , f y d de las direcciones de memoria indican el número de la página lógica, el marco físico y el desplazamiento dentro de éstos, respectivamente. La descomposición es la siguiente:

Dirección de memoria lógica o virtual: $p + d$ bits

Número de página: Campo de longitud p (bits más significativos)	Desplazamiento dentro de la página: Campo de longitud d (bits menos significativos)
--	--

Dirección de memoria física: $f + d$ bits

Número de marco (<i>frame</i>): Campo de longitud f (bits más significativos)	Desplazamiento dentro del marco: Campo de longitud d (bits menos significativos)
--	---

Numéricamente, los campos se corresponden con el logaritmo binario de los respectivos tamaños máximos que se pueden representar en cada caso, así que expresando las cantidades en potencias de dos veremos todo más claro:

- La página de memoria es de 2^{12} palabras de un byte, por lo que el campo desplazamiento dentro de la página tiene una longitud de $d = 12$ bits. Este desplazamiento sirve también dentro del marco.
 - La memoria física tiene 2^{35} palabras de un byte, por lo que sus direcciones físicas tienen una longitud de 35 bits. De ellos, los 12 bits menos significativos corresponden al desplazamiento d , y los 23 restantes indicarán el marco de memoria física, concluyendo que $f = 23$ bits.
 - La memoria virtual tiene 2^{46} palabras de un byte, por lo que sus direcciones lógicas se representan numéricamente con 46 bits. De forma similar al caso anterior, los 12 bits menos significativos son para el desplazamiento dentro de la página, quedando los $p = 34$ bits superiores para indicar la página de memoria en la que se encuentra dicha dirección.

A partir de ahí, la traducción de cada dirección se realiza resolviendo el campo f que corresponde a cada valor de p , y conservando el mismo desplazamiento d en ambos casos, que se concatena a continuación si todo lo expresamos en bits.

Por otro lado, la TLB guarda las correspondencias entre **p** y **f** para las traducciones que han sido resueltas de forma más reciente y/o frecuente, por lo que sus ocho entradas almacenarán cada una 59 bits en total: los 34 del campo **p** y los 23 del campo **f** que indican el marco de memoria física que aloja los datos de esa página. Esos campos no se registran como una tabla cualquiera dentro de la TLB: la columna correspondiente al campo **p** se implementa con una memoria asociativa en la que la *MMU (Memory Management Unit)* consulta todos los valores en paralelo y por contenido, con objeto de resolver de forma instantánea la traducción en caso de que ésta se encuentre en una de las entradas de la TLB y devolver el valor del campo **f** de forma inmediata. Recordemos que el único objetivo de una TLB es acelerar el proceso de traducción, por lo que si éste no se optimiza, su concurso no aporta nada.

8. Sobre el PC del ejercicio anterior, en un instante determinado la TLB tiene los contenidos que se muestran a continuación:

TIB

entrada	página	marco
0	0001000000000100000000010000000001	011111111111111111111110
1	111111111111111111111111111111111111	00000000000000000000000000000000
2	00000000000000000000000000000000000000	111111111111111111111111
3	11101111111101111111101111111110	10000000000000000000000001

Las entradas de la TLB se van llenando de arriba a abajo, y se reemplazan con algoritmo FIFO. Si una entrada abandona la TLB, consideraremos que conserva sus valores si vuelve a entrar (es decir, la traducción de página a marco no ha cambiado mientras estuvo fuera de la TLB).

Sea la siguiente secuencia de direcciones de acceso a memoria solicitadas desde un proceso:

Con todos estos elementos, se pide lo siguiente:

- a) Indicar la secuencia de aciertos/fallos a TLB para cada una de las peticiones a memoria, mostrando la traducción de dirección virtual a física en el caso de los aciertos a TLB. Puedes ayudarte de la tabla adjunta para ilustrar todo el proceso.

	Petición de memoria (4 bits más altos)	¿Acierto en TLB? (Sí ^(*) / No)	¿A qué entrada de la TLB reemplaza? (posición y 4 bits más altos)	¿Cuál es la dirección física resultante al traducir? (colocar marco y desplazamiento)
1	1111...			
2	0000...			
3	0000...			
4	1111...			
5	1010...			
6	0101...			
7	1111...			
8	1111...			
9	0000...			

(*) En caso de acierto, indica entre paréntesis el número de entrada de la TLB en la que se produce.

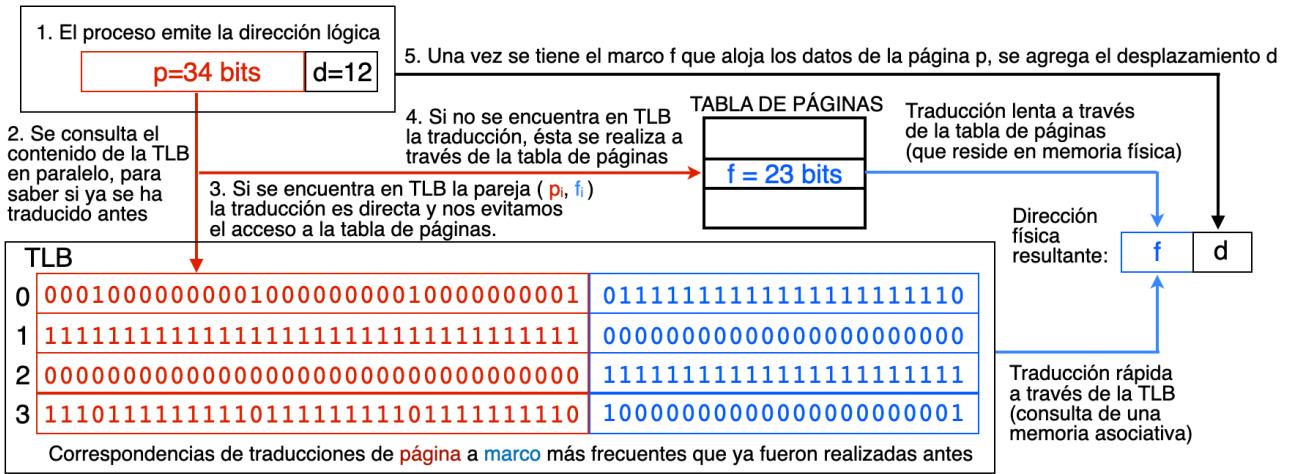
- b) Si la TLB utiliza el algoritmo LRU para reemplazar las traducciones menos usadas recientemente, muestra los resultados en otra tabla similar, comparando el resultado de esta opción frente a la anterior (en la que el reemplazo era FIFO).
 - c) La TLB es tan importante para mejorar el proceso de traducción de direcciones, que cada CPU lleva una integrada. En el caso del procesador i9 de Intel, su TLB dispone de 32 entradas de 4 vías (4-way). Estas vías se corresponden con el número de entradas que hay en conjuntos (sets) predefinidos algebráicamente de forma disjunta a través de sistemas de congruencias (operaciones módulo de una división para las direcciones de memoria). De esta manera, las consultas a TLB para conocer si una traducción ya se ha realizado antes y puede amortizarse, tienen lugar dentro del conjunto, lo que agiliza notablemente la respuesta de la TLB. Puesto que hay 32 entradas y 4 vías, resultan 8 conjuntos establecidos como sigue:

- Si al dividir la dirección por 8 sale resto 0, la traducción se guarda en el conjunto 0.
 - Si al dividir por 8 resulta resto 1, se guarda en el conjunto 1.
 - ... y así sucesivamente hasta concluir con el conjunto 7.

Al dirigirnos al conjunto de forma directa mediante esa simple operación módulo y limitar la búsqueda dentro de él, no sólo aceleramos el proceso, sino también el coste de la memoria asociativa necesaria para consultar por contenido. Por el lado negativo, se asume el riesgo de que muchas peticiones pudieran coincidir en un mismo conjunto, aumentando el número de reemplazos (aunque esto es algo improbable según la ley estadística de los grandes números). Atrévete a experimentar aquí con esta idea, considerando una sencilla TLB de sólo 4 entradas organizadas en dos conjuntos de 2 vías: Las direcciones pares van al conjunto 0 y rivalizan dentro de él (aplica reemplazo FIFO en caso de ser necesario), y las direcciones impares

van al conjunto 1, compitiendo por sus dos entradas de forma similar. Si nos fijamos bien en la figura anterior que mostraba los contenidos de nuestra TLB, el conjunto 1 estaría compuesto por la mitad superior de la TLB (sus dos entradas contienen números impares), y el conjunto 0 estaría en la mitad inferior (con los números de dirección par). Puedes construir una tabla similar a las anteriores, incorporando una nueva columna que refleje el conjunto al que pertenece cada petición para gestionar los posibles reemplazos dentro de él.

A continuación resumimos la forma de proceder para traducir las direcciones virtuales a físicas cuando la TLB apoya a la tabla de páginas:

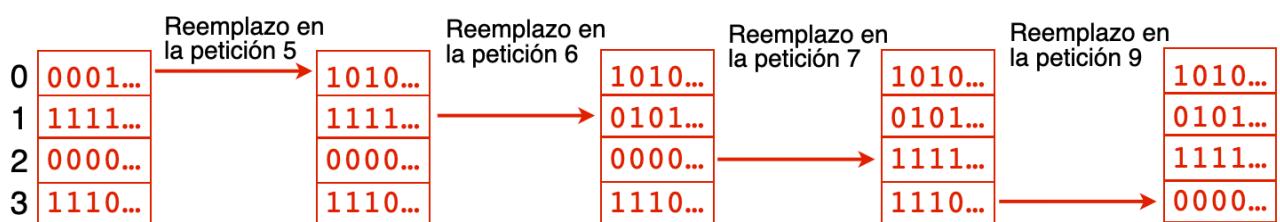


Apartado a)

Aplicando el algoritmo de reemplazo FIFO dentro de la TLB cuando no se produce un acierto, la tabla se completa de la siguiente forma:

Petición de memoria	¿Acierto en TLB?	¿A qué entrada de la TLB reemplaza?	Dirección física traducida si se acierta en la TLB (marco y desplazamiento)
1 1111...	Sí (en 1)	A ninguna	00000000000000000000000000000000111111111111
2 0000...	Sí (en 2)	A ninguna	1111111111111111111111111000000000000
3 0000...	Sí (en 2)	A ninguna	1111111111111111111111111001111111111
4 1111...	Sí (en 1)	A ninguna	000000000000000000000000000000001100000000000
5 1010...	No	A la 0, 0001...	
6 0101...	No	A la 1, 1111...	
7 1111...	No	A la 2, 0000...	
8 1111...	Sí (en 2)	Ninguna	00000000000000000000000000000000111111111111
9 0000...	No	A la 3, 1110...	

Y la secuencia de 4 reemplazos FIFO en TLB es la siguiente:

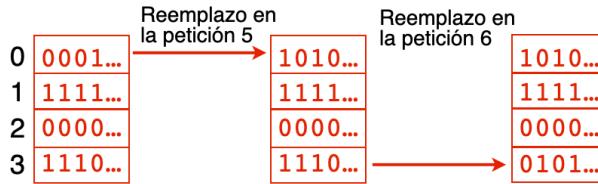


Apartado b)

Cuando la TLB reemplaza por LRU (en lugar de FIFO), la evolución es la siguiente:

	Petición de memoria	¿Acierto en TLB?	¿A qué entrada de la TLB reemplaza?	Dirección física traducida si se acierta en la TLB (marco y desplazamiento)
1	1111...	Sí (en la 1)	A ninguna	00000000000000000000000000000000111111111111
2	0000...	Sí (en la 2)	A ninguna	11111111111111111111111111111111000000000000
3	0000...	Sí (en la 2)	A ninguna	11111111111111111111111111111111001111111111
4	1111...	Sí (en la 1)	A ninguna	000000000000000000000000000000001100000000000
5	1010...	No	A la 0, 0001...	
6	0101...	No	A la 3, 1110...	
7	1111...	Sí (en la 1)	Ninguna	00000000000000000000000000000000111111111111
8	1111...	Sí (en la 1)	Ninguna	00000000000000000000000000000000111111111111
9	0000...	Sí (en la 2)	Ninguna	11111111111111111111111111111111001111111111

Y la secuencia de reemplazos LRU en TLB es más breve, pues se producen sólo dos:



La participación de la TLB nos ahorra, por tanto, cinco accesos a la tabla de páginas, que suben a siete si adoptamos el algoritmo LRU en lugar del FIFO. En total, sólo necesitamos dos consultas a la tabla de páginas para traducir la secuencia de nueve referencias a memoria.

Apartado c)

Aquí la TLB dispone de dos conjuntos de dos vías y reemplaza por FIFO (siempre dentro del conjunto, por lo que tanto en caso de acierto como en caso de fallo esta vez indicaremos el número de la entrada dentro de éste). En total se producen tres reemplazos, tal y como indicamos a continuación:

	Petición de memoria	¿Conjunto?	¿Acierto en TLB?	Entrada TLB reemplazada	Dirección física traducida si se acierta en la TLB (marco y desplazamiento)
1	1111...	1	Sí (en la 1)	Ninguna	00000000000000000000000000000000111111111111
2	0000...	0	Sí (en la 0)	Ninguna	11111111111111111111111111111111000000000000
3	0000...	0	Sí (en la 0)	Ninguna	11111111111111111111111111111111001111111111
4	1111...	1	Sí (en la 1)	Ninguna	000000000000000000000000000000001100000000000
5	1010...	0	No	0. 0000...	
6	0101...	1	No	0. 0001...	
7	1111...	1	Sí (en la 1)	Ninguna	00000000000000000000000000000000111111111111
8	1111...	1	Sí (en la 1)	Ninguna	00000000000000000000000000000000111111111111
9	0000...	0	No	1. 1110...	

La secuencia de 3 reemplazos FIFO para el caso de 2 conjuntos de 2 entradas cada uno es la siguiente:



Finalmente, y sólo como curiosidad, si la TLB de dos conjuntos de dos vías reemplazase por LRU en lugar de FIFO, puedes comprobar que se producen los mismos dos reemplazos que en la TLB convencional con reemplazo LRU (apartado b). Así que en este caso lograríamos una aceleración en las consultas a TLB sin reducir el número de aciertos. Con el reemplazo FIFO, hemos visto que la TLB de dos conjuntos sale mejor parada respecto a la TLB convencional, pues el número de reemplazos a TLB se reduce de cuatro a tres, y además los conjuntos agilizan las consultas a dicha TLB.

9. El sistema de memoria anterior, con los mismos valores iniciales ya expuestos para la TLB, tiene en ese mismo instante el contenido de la tabla de páginas y la memoria física que se indica a continuación:

Memoria física (DRAM)			
	Marco	Desplaz.	Dato
0	0	0	a
0	1
0	...	4095	v
1	0	0	b
1	1
1	...	4095	x
...	0
...	1
...	...	4095	...
2 ³⁴ -2	0	0	c
2 ³⁴ -2	1
2 ³⁴ -2	...	4095	y
2 ³⁴ -1	0	0	d
2 ³⁴ -1	1
2 ³⁴ -1	...	4095	z

Tabla de páginas

0	11111111111111111111111111111111
1	11111111111111111111111111111110
...
...
...
2 ³⁴ -2	00000000000000000000000000000001
2 ³⁴ -1	00000000000000000000000000000000

Se detallan únicamente las dos primeras y las dos últimas entradas de la tabla de páginas, cuyo valor es un número de marco de 23 bits. Por otra parte, mostramos los valores de la memoria física a la derecha, indicando el dato almacenado en la primera y última posición de algunos marcos (no es necesario conocer más datos para resolver el ejercicio en su totalidad, lo cual es una pista que puede ayudarnos mucho en su resolución). Al ser una memoria direccionable a nivel de byte, los datos que contiene se han considerado valores de tipo `char` del lenguaje C, reflejando cada posición de memoria un carácter ASCII diferente (a, v, b, x, ...). Esto nos permite diferenciar el contenido de cada palabra de memoria.

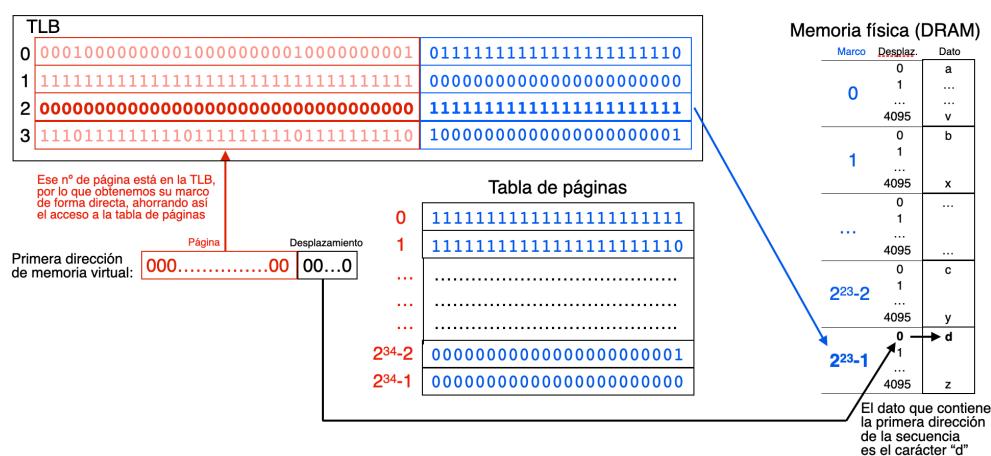
En esta situación, un programa solicita la siguiente secuencia de cuatro direcciones de memoria virtual:

Se pide encontrar el dato almacenado en cada una de esas cuatro posiciones de memoria, teniendo en cuenta tanto la consulta a la TLB como a la tabla de páginas cuando sea necesario en cada caso.

Para la primera dirección, su campo p que indica la página lógica está compuesto por 34 ceros, y su desplazamiento d por 12 ceros.

- Lo primero que hacemos es consultar en paralelo el valor de las 4 entradas rojas de la TLB, para saber si el campo p coincide con alguna de ellas.
 - En este caso, coincide con la tercera entrada, por lo que el campo azul adyacente nos proporciona directamente el marco de memoria física donde se encuentran ubicados los datos correspondientes a esta página (que ya habría sido solicitada en un pasado muy reciente, y por ello la TLB retuvo aquí su traducción, lo que nos permite ahora reutilizarla).
 - El marco almacenado en la tercera entrada de la TLB es un campo **f** de 23 bits cuyo valor resulta ser de 23 unos. Por lo tanto, esta página está en el marco numéricamente más alto, esto es, el último de todos.
 - Dentro del marco anterior, usamos ahora el campo desplazamiento, **d**, para localizar el dato asociado a esa dirección. Este campo **d** tiene 12 bits, que resultan ser todos cero, por lo que se está solicitando el primer dato de ese marco, cuyo valor es el carácter “d”.

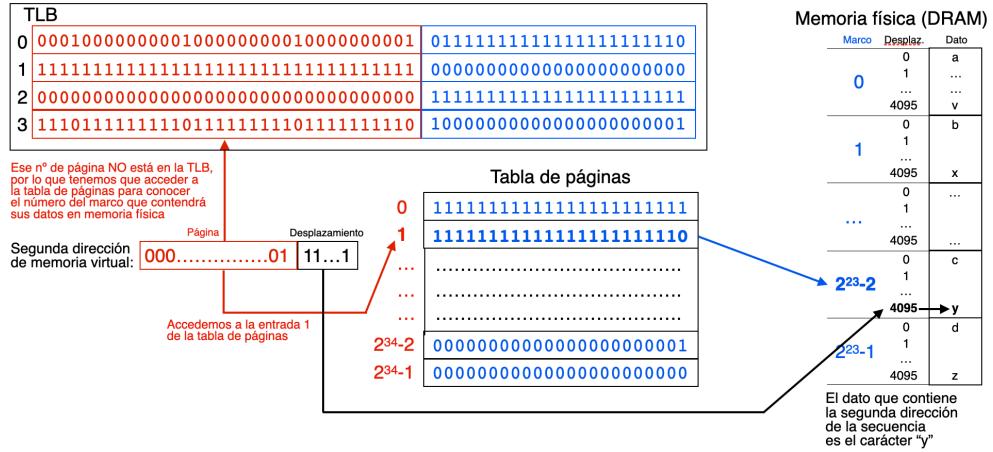
El proceso completo de traducción puede resumirse en el siguiente diagrama:



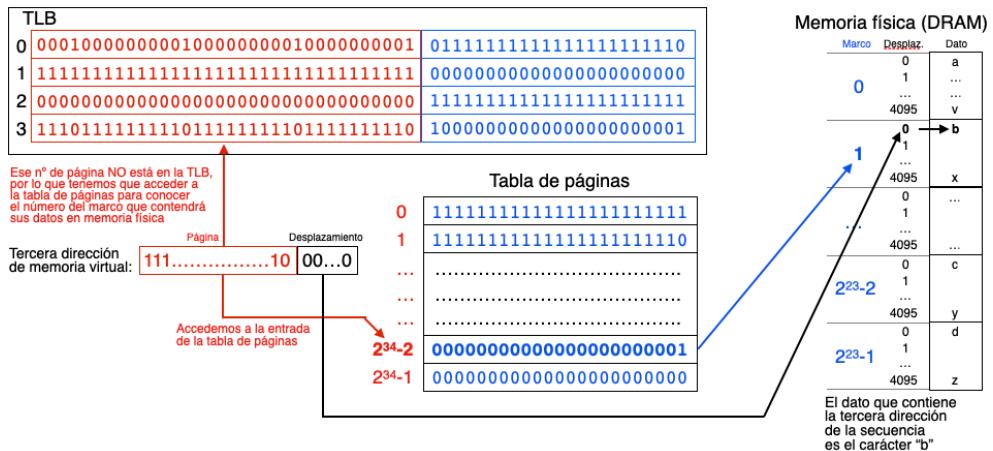
Para la segunda dirección, su campo p que indica la página lógica está compuesto por 33 ceros seguidos de un uno en la posición menos significativa, y su desplazamiento d por 12 unos.

- Consultamos la TLB, y en este segundo caso no tenemos tanta fortuna: Esa traducción no ha sido retenida por la TLB, así que hay que acceder a la tabla de páginas para conocer el marco asociado a esa página.
 - A partir de ese marco, el campo desplazamiento, d, nos permite acceder a la dirección relativa al comienzo de ese marco. Los 12 unos de ese campo d indican que esta dirección es la última dentro del marco, en este caso la 4095, cuyo valor es el carácter “y”.

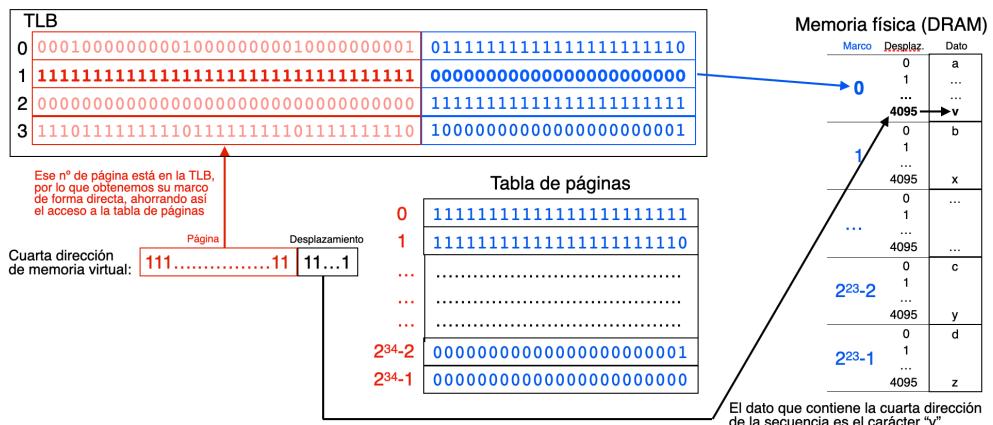
Para esta segunda dirección, el proceso de traducción se resume en el siguiente diagrama:



La tercera dirección virtual tampoco está en la TLB, por lo que el procedimiento para su traducción es análogo al de la segunda dirección virtual, que resumimos a continuación:



Por último, la cuarta dirección virtual sí acierta cuando consulta la TLB, por lo que el proceso de traducción se asemeja al de la primera dirección virtual, quedando como sigue:



Por lo tanto, los datos almacenados en las cuatro posiciones de memoria virtual indicadas en la secuencia que se nos solicita son los caracteres "d", "y", "b" y "v", por ese orden.

10. Hasta este punto hemos manejado volúmenes y direcciones de memoria realistas en el rango de los Terabytes, con la consiguiente dificultad para manejarlos con números que superan los 40 bits de longitud. En este último ejercicio usaremos direcciones más cortas que nos permitan estudiar con sencillez el alojamiento y la fragmentación de la memoria. Considera para ello un espacio físico de memoria de tan sólo 64 Kbytes con páginas de 4 Kbytes, y un espacio lógico para cada proceso de 128 Kbytes con segmentación paginada. En un momento dado, consideremos que sólo 40 Kbytes de esa memoria se encuentran ocupados por un único proceso, cuya tabla de páginas adjunta revela el uso que hace de la memoria.

Tabla de páginas	
0	0001
1	0110
2	0000
3	0100
4	1111
5	0010
6	0101
7	1011
8	0111
9	1100

Se pide lo siguiente:

- a) Ilustra cómo asignaría el Sistema Operativo los restantes 24 Kbytes a un programa que ocupe 22 Kbytes divididos en sus 3 segmentos típicos:

- Segmento de código: 5 Kbytes.
- Segmento de datos: 10 Kbytes.
- Segmento de pila: 7 Kbytes.

Considera que se asigna primero el marco de memoria libre numéricamente más bajo, y que los segmentos se alojan en el orden (1) código, (2) datos y (3) pila.

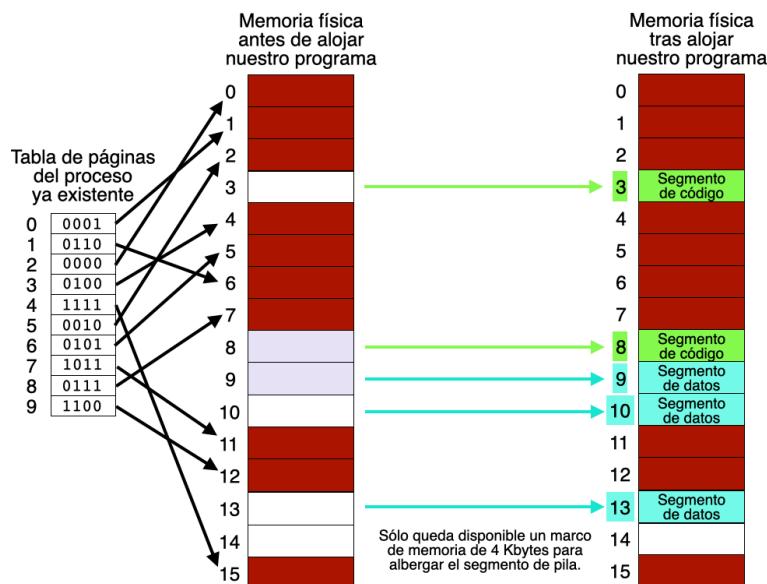
- b) En realidad, el programa sólo necesita alojar su segmento de código y su segmento de datos para echar a andar (los delimita con antelación el compilador), mientras que el segmento de pila arranca desde cero y va creciendo en tiempo de ejecución según se llama a funciones y librerías. Más realista resulta, por tanto, no asignar un tamaño fijo a este segmento de pila, y considerar que crecerá hasta donde le permita el Sistema Operativo. Olvídate por tanto del tamaño de 7 Kbytes dado para este último segmento e indica cuál sería el máximo tamaño que podría ocupar durante la ejecución del programa si el Sistema Operativo sólo le dejara crecer mientras quede memoria física libre.

- c) Indica cuánto espacio se desperdicia en el caso anterior por fragmentación interna y externa.
- d) Compara las fragmentación anterior con el caso de que redujéramos el tamaño de página a la mitad (es decir, 2 Kbytes) para tu programa (el proceso que ya ocupaba la memoria conserva el mismo espacio que tenía asignado). ¿Qué tamaño máximo podría ocupar el segmento de pila en este caso?
- e) Supongamos que el sistema de memoria no está paginado, y que por tanto la asignación de memoria debe hacerse de forma contigua. Tomando el mismo punto de partida (40 Kbytes de memoria ocupados por el proceso inicial y 24 Kbytes libres que ahora no están organizados en marcos), indica cómo llenarían la memoria los tres segmentos alojándose en el orden código-datos-pila, con tamaños de 5, 10 y 4 Kbytes, respectivamente. Considera primero *best-fit* como algoritmo para el alojamiento de memoria no contigua, e ilustra después las diferencias respecto a *first-fit* y *worst-fit*.
- f) ¿Hasta qué tamaño podría crecer la pila en tiempo de ejecución en este sistema de memoria no contigua? (toma como referencia el alojamiento con *best-fit*)

- g) ¿Qué algoritmo(s) de los tres estudiados (*best-fit*, *first-fit* y *worst-fit*) no consigue(n) alojar completamente en memoria los tres segmentos de tu programa? ¿qué fenómeno adverso sufre el Sistema Operativo para no poder completar el alojamiento de tu programa aún existiendo memoria suficiente en los 24 Kbytes de memoria disponibles?

- a) La tabla de páginas del proceso existente revela el uso de la memoria que mostramos a la izquierda en el diagrama anexo, y que corresponde al punto de partida para resolver el ejercicio. A partir de ahí, los tres segmentos de tu programa se alojan en el orden código-datos-pila, por lo que la secuencia de ocupación de los marcos de memoria es la siguiente:

- 1) El segmento de código necesita dos marcos de 4 Kbytes para alojar sus 5 Kbytes. Ocupará los marcos 3 y 8, que son los primeros disponibles.
- 2) El segmento de datos necesita tres marcos para alojar sus 10 Kbytes, que serán los números 9, 10 y 13.
- 3) El segmento de pila no puede alojarse en su totalidad, ya que tiene 7 Kbytes y sólo queda disponible el marco 14, cuyo tamaño es de 4 Kbytes.

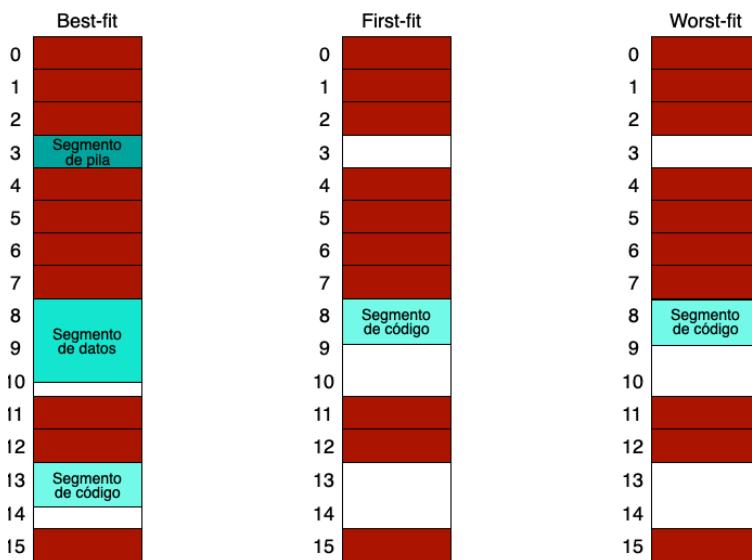


- b) El máximo espacio que puede ocupar el segmento de pila es de 4 Kbytes, una vez que el segmento de código ha ocupado 2 marcos y el de datos los 3 marcos siguientes, tal y como muestra la parte inferior derecha del diagrama anterior.
- c) El segmento de código ocupa 8 Kbytes (2 marcos) pero sólo necesita 5 Kbytes, por lo que desperdicia 3 Kbytes por fragmentación interna de los 4 Kbytes que le proporciona su segundo marco. De forma similar, el segmento de datos ocupa 12 Kbytes (3 marcos) y desperdicia 2 Kbytes en su tercer marco por fragmentación interna. En total, se pierden $3+2=5$ Kbytes de memoria, que son los que evitan que el segmento de pila de 7 Kbytes considerado inicialmente pueda alojarse en memoria. En este caso, no entra en juego la fragmentación externa, que tiene lugar cuando el sistema de memoria no está paginado.
- d) Si reducimos el tamaño de página de 4 Kbytes a 2 Kbytes:
- El segmento de código ocuparía 6 Kbytes (3 marcos, con pérdida de 1 Kbyte en el último marco por fragmentación interna).
 - El segmento de datos ocuparía 10 Kbytes (5 marcos completos).

- El segmento de pila podría ocupar hasta 8 Kbytes (4 marcos de 2 Kbytes), puesto que el total disponible son 24 Kbytes (12 marcos, y sólo hemos ocupado $3+5=8$).

En este caso, sólo se desperdiciaría 1 Kbyte por fragmentación interna, que se produce en el último marco del segmento de código. Los otros dos segmentos ocupan exactamente un múltiplo del tamaño de página, y por lo tanto, aprovechan íntegramente su memoria.

- El único algoritmo que consigue alojar los tres segmentos en memoria es *best-fit*, que busca entre todos los huecos disponibles aquel que mejor se ajusta al tamaño de cada segmento. Tanto *first-fit* como *worst-fit* asignan al segmento de código el bloque de 12 Kbytes, y como es el único que puede albergar el segmento de datos de 10 Kbytes, cuando éste llega después, no encuentra un hueco suficientemente grande para alojarse. La siguiente ilustración ubica todos los alojamientos que pueden completarse en cada caso.



- El segmento de pila no puede ocupar más espacio de los 4 Kbytes que ya tiene asignados con *best-fit*, pues corresponde exactamente al hueco que tiene asignado.
- La fragmentación externa que sufre el alojamiento contiguo de memoria es lo que impide que, aunque exista más espacio disponible del que se solicita, no podamos alojar los tres segmentos del programa usando los algoritmos *first-fit* o *worst-fit*. Habría que realizar una compactación de los huecos (fusionar previamente todos ellos en un solo bloque) para que esto fuese posible.



Ejercicios de Sistemas Operativos. Tema 4: Sistemas de ficheros

Ingeniería Informática. Grupo A

Profesor: Manuel Ujaldón

Dpto. de Arquitectura de Computadores. UMA



1. Un disco duro gira a una velocidad angular constante de 1000 RPM (revoluciones por minuto). Calcular su latencia rotacional, es decir, el tiempo máximo que tardará en encontrar un dato en una pista determinada una vez que el cabezal se encuentre ubicado en esa pista.

El caso más desfavorable es que el sector de la pista acabe justo de pasar tras el cabezal, en cuyo caso, tardará el tiempo necesario para dar una vuelta completa. Dado que gira a 1000 revoluciones por minuto, completa 1000 vueltas en 60 segundos, es decir, tarda 60 milisegundos en dar una vuelta.

2. Calcula la tasa de transferencia o ancho de banda máximo en Mbytes/sg. de un disco duro que gira a 7200 RPM y tiene 16 sectores de 8 Kbytes en cada pista.

El disco recorre 7200 pistas por minuto, y cada pista alberga 16 sectores de 8 Kbytes, o sea, 128 Kbytes de datos. Por tanto, se leen 7200×128 Kbytes / minuto = 921.600 Kbytes / minuto = 15 Mbytes / sg.

3. Un disco duro se compone de 32 cabezales (*heads*), 16 platos (*platters*), 25600 cilindros (*cylinders*) y 128 sectores por pista, con un tamaño de sector de 8 Kbytes (suponer que coinciden el tamaño del sector y del *cluster*). Se pide calcular la capacidad del disco y el espacio ocupado por una FAT32.

Cada plato tiene dos superficies de lectura, el envés y el revés, por lo que para 16 platos son necesarios los 32 cabezales descritos. Por otro lado, cada cilindro consta de 32 pistas paralelas situadas a diferentes alturas (una para cada cabezal de ese cilindro). Con todo esto, tenemos:

Capacidad de disco = 32 cabezales × 25600 pistas/cabezal × 128 sectores/pista × 8 Kbytes/sector = $2^5 \times 2^8 \times 100 \times 2^7 \times 2^{13}$ bytes = 800 Gbytes.

Respecto al espacio ocupado por la FAT32, ésta albergará un puntero de 32 bits a cada sector. El número total de sectores es de 32 cabezales × 25600 pistas/cabezal × 128 sectores/pista = 100 Msectores. Por lo tanto, se necesitan 100 Msectores × 32 bits = 3200 Mbits = 400 Mbytes.

Asumiendo que en el disco hay algunos metadatos de gestión adicionales a la FAT de esta partición, el espacio útil de disco para esta partición única estaría en torno a los 799.5 Gbytes.

4. Un Sistema Operativo utiliza clusters de 1 Kbyte y una FAT12 para gestionar el almacenamiento de un disco duro. ¿Qué capacidad máxima puede tener cada partición de ese disco? ¿Y si se trata de una FAT32? ¿Cuánto ocupa la FAT en cada caso?

La capacidad máxima para una FAT12 es de 2^{12} clusters. Puesto que en este caso los clusters son de 1 Kbyte, esta capacidad sería de $2^{12} \times 1$ Kbyte = 4 Mbytes.

Para una FAT32, la capacidad máxima es de 2^{32} clusters, esto es, $2^{32} \times 1$ Kbyte = 4 Tbytes.

La FAT12 ocupará 2^{12} entradas de 12 bits, es decir, 6 Kbytes.

La FAT32 ocupará 2^{32} entradas de 32 bits, es decir, 16 Gbytes.

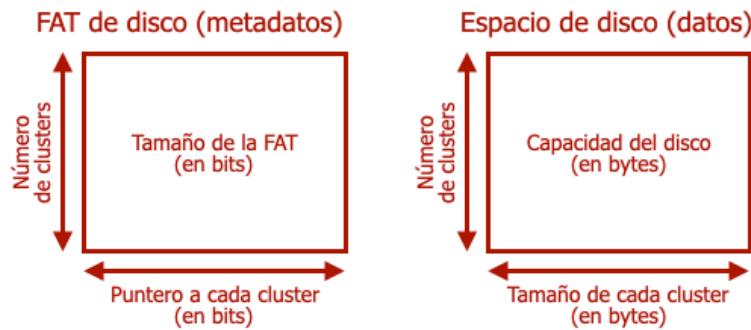
5. Sea un disco de 4 Terabytes formateado en FAT32. ¿Cuál es el tamaño mínimo del *cluster*? ¿Qué espacio ocupa la FAT32?

Una FAT32 emplea 32 bits en el direccionamiento a los clusters, de ahí que 2^{32} clusters deben cubrir al menos los 4 Terabytes del espacio de disco. Esto nos conduce a la siguiente expresión: $2^{32} \text{ clusters} \times (\text{Tamaño del cluster}) \geq 4 \text{ Terabytes}$. Despejando tenemos:

$$\text{Tamaño del cluster} \geq \frac{4 \text{ Terabytes}}{2^{32}} = \frac{2^{42} \text{ bytes}}{2^{32}} = 2^{10} \text{ bytes} = 1 \text{ Kbyte.}$$

Para calcular el espacio ocupado por la FAT32, sabemos que ésta aloja 2^{32} entradas de 32 bits cada una, y por lo tanto, ocupa $2^{32} \text{ entradas} \times 32 \text{ bits/entrada} = 2^{32} \times 4 \text{ bytes} = 16 \text{ Gbytes}$.

Las siguientes figuras pueden ayudarnos a establecer visualmente las dos relaciones que debemos tener claras para resolver este ejercicio.



6. Sea un disco de 2 Terabytes formateado en FAT32 con clusters de 64 Kbytes. Se pide:

- a) Calcular el número de clusters del disco.

Si dividimos el espacio total de 2 Terabytes entre el tamaño que ocupa cada cluster, obtendremos el número de clusters del disco:

$$\frac{2 \text{ Tbytes}}{64 \text{ Kbytes}} = \frac{2^{41} \text{ bytes}}{2^{16} \text{ bytes}} = 2^{25} \text{ clusters} = 32 \text{ Mclusters tiene el disco.}$$

- b) Calcular el tamaño de la FAT32.

La FAT tiene tantas entradas como clusters tiene el disco, es decir, 2^{25} entradas. Cada entrada ocupa 32 bits (4 bytes), por lo que la FAT ocupa $2^{25} \text{ entradas} \times 4 \text{ bytes/entrada} = 2^{27} \text{ bytes} = 128 \text{ Mbytes}$.

- c) ¿Cuál es el número máximo de archivos que podemos tener en ese disco?

El disco puede alojar tantos ficheros como entradas tiene la FAT, siempre que cada fichero ocupe un cluster como máximo. Por lo tanto, serían un total de 128 Mficheros, o $128 \times 1024 \times 1024$ ficheros. Notar que en este caso, todas las entradas de la FAT tendrían un valor NULL, puesto que no hay ninguna cadena de clusters que construir dentro de ella. El puntero que localiza el cluster que ocupa cada fichero no vendría dado por la FAT32, sino por la cabecera de dicho fichero junto a otros campos como el nombre del fichero y sus permisos.

- d) ¿Qué directorio sufriría un mayor impacto de la fragmentación interna: uno que contiene 50 ficheros de 1 Kbyte u otro que contiene 100 ficheros de 2 Kbytes?

Cada fichero debe alojar un cluster de datos, que en este caso es de 64 Kbytes. Si necesita menos tamaño, como es el caso, el resto se pierde por fragmentación interna. Por lo tanto:

- Para el primer directorio, cada uno de los 50 ficheros desperdician 63 Kbytes, y en total se pierden $50 \times 63 \text{ Kbytes} = 3150 \text{ Kbytes de disco}$.
- Para el segundo directorio, cada uno de los 100 ficheros desperdician 62 Kbytes, perdiéndose en total $100 \times 62 \text{ Kbytes} = 6200 \text{ Kbytes de disco}$.

En consecuencia, el segundo directorio desperdicia más espacio por fragmentación interna.

7. Un Sistema Operativo UNIX utiliza clusters de 2 Kbytes y punteros de 32 bits para direccionarlos, gestionando el espacio de almacenamiento para cada archivo en disco mediante el típico i-nodo que registra 10 punteros directos, un puntero indirecto simple, un puntero indirecto doble y otro triple. Responder a las siguientes cuestiones:

- a) ¿Cuántos punteros de clusters caben en un cluster de disco?

El *cluster* es de 2 Kbytes y los punteros son de 32 bits (4 bytes). Por lo tanto, en un *cluster* caben $\frac{2 \text{ Kbytes}}{4 \text{ bytes}} = 512$ punteros.

- b) ¿Cuál es el máximo tamaño de fichero soportado?

Con los 10 punteros directos podemos direccionar hasta $10 \times 2 \text{ Kbytes} = 20 \text{ Kbytes}$.

Dado que en un *cluster* caben 512 punteros a *clusters*, con el puntero indirecto simple (PIS) podemos direccionar hasta $512 \text{ clusters} \times 2 \text{ Kbytes/cluster} = 1 \text{ Mbyte}$.

El puntero indirecto doble (PID) dirige a 512 PIS × 1 Mbyte/PIS = 512 Mbytes.

El puntero indirecto triple (PIT) dirige a 512 PID × 512 Mbytes/PID = 256 Gbytes.

En total, el espacio máximo de disco sería 256 Gbytes + 512 Mbytes + 1 Mbyte + 20 Kbytes.

- c) ¿Cuántos punteros indirectos deben habilitarse para albergar un fichero de 20 Mbytes?

Para albergar un fichero de 20 Mbytes hace falta habilitar hasta el PID, ya que con él llegamos hasta 512 Mbytes, mientras que con el PIS sólo llegamos a 1 Mbyte.

8. La organización de un Sistema Operativo Linux utiliza i-nodos con 16 punteros directos a *clusters* de 8 Kbytes, un puntero indirecto simple, otro doble y otro triple. Los punteros son de 32 bits, de los cuales 8 tienen información de control para la partición y los 24 bits restantes se utilizan para direccionar al *cluster*. Se pide:

- a) ¿Cuál es el máximo tamaño de fichero soportado?

Con los 16 punteros directos podemos direccionar hasta $16 \times 8 \text{ Kbytes} = 128 \text{ Kbytes}$.

Por otro lado, en un *cluster* de 8 Kbytes caben 8 Kbytes / 4 bytes = 2048 punteros.

El puntero indirecto simple (PIS) referencia a un *cluster* lleno de punteros, es decir, contendrá 2048 punteros para direccionar hasta $2048 \text{ clusters} \times 8 \text{ Kbytes/cluster} = 16 \text{ Mbytes}$.

El puntero indirecto doble (PID) dirige a 2048 PIS × 16 Mbyte/PIS = 32 Gbytes.

El puntero indirecto triple (PIT) dirige a 2048 PID × 32 Gbytes/PID = 64 Tbytes.

En total, el espacio máximo de disco sería 64 Tbytes + 32 Gbytes + 16 Mbytes + 128 Kbytes

- b) ¿Cuál es el máximo tamaño de partición soportado?

Con 24 bits de dirección para el *cluster* se pueden gestionar hasta 2^{24} *clusters* de 8 Kbytes. Multiplicando ambas cifras, obtenemos que 128 Gbytes es el máximo tamaño de una partición.

- c) ¿Puede ser el límite máximo para la partición inferior al límite máximo para cada uno de sus ficheros? ¿Por qué? ¿Cuál es el tamaño máximo de fichero que puede abarcarse al incluir en el i-nodo el puntero indirecto triple?

La partición siempre será más grande que los ficheros, puesto que alberga a todos ellos. Lo que ocurre aquí es que para una partición de 128 Gbytes no queremos limitar el máximo tamaño de archivo a 32 Gbytes, que es lo que se alcanza con el puntero indirecto doble. Habilitando el puntero indirecto triple para superar esta cota, el direccionamiento a disco

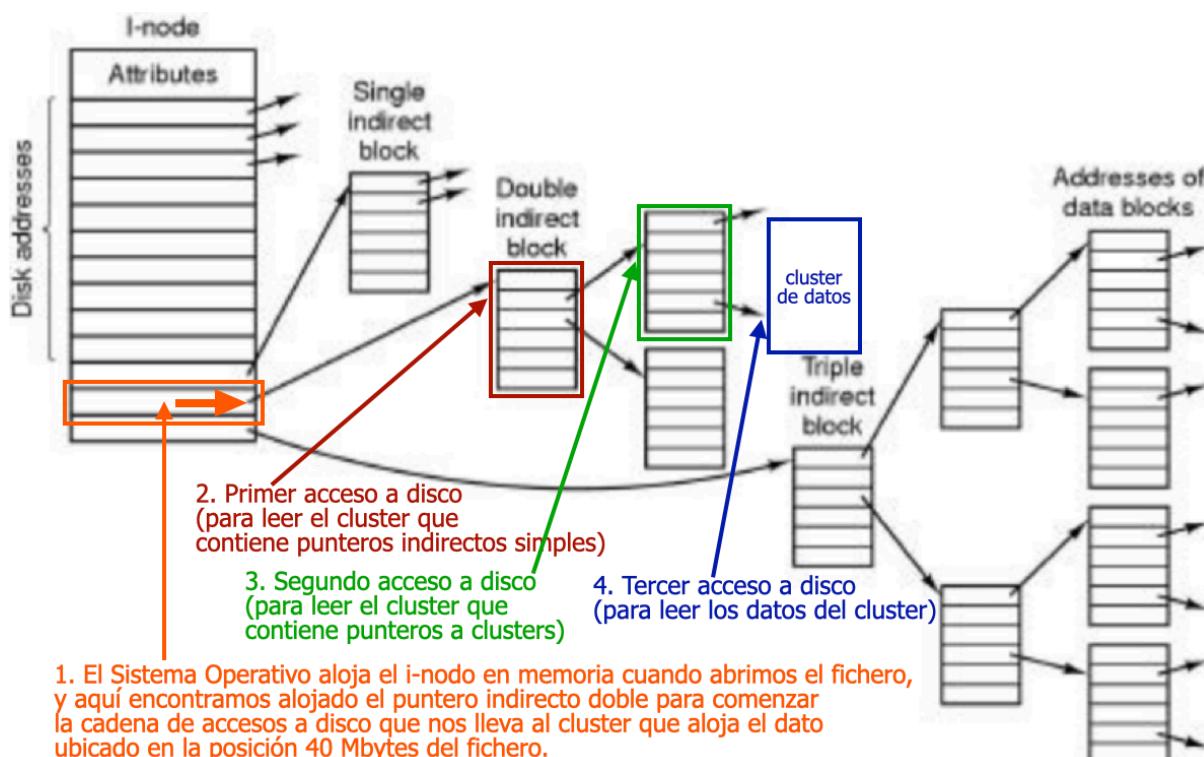
puede alcanzar hasta 64 Tbytes, pero en esta partición desaprovecharíamos buena parte de este rango, limitándonos a cubrir con el puntero indirecto triple el rango de datos que comienza en los 32 Gbytes y concluye en los 128 Gbytes, donde finaliza la partición. Para apurar todos esos 128 Gbytes, el fichero tendría que ser el único del sistema, lo que sugiere prescindir del puntero indirecto triple dentro del i-nodo y limitar el fichero a los 32 Gbytes.

- d) ¿Cuántos accesos a disco son necesarios si quisieramos acceder al byte 41.943.040 del fichero? (ese número es $40 \times 1024 \times 1024$, es decir, 40 Mbytes)

De la solución de los apartados anteriores sabemos que con los 16 punteros directos alcanzamos a direccionar los primeros 128 Kbytes del fichero, y con el puntero indirecto simple llegamos hasta los 16 Mbytes. Por lo tanto, es necesario habilitar el puntero indirecto doble para acceder al dato alojado en la posición 40 Mbytes (para que hubiera hecho falta el puntero indirecto triple tendrían que habernos solicitado un byte por encima de los 32 Gbytes + 16 Mbytes + 128 Kbytes). Si suponemos que el i-nodo de ese fichero se carga en memoria cuando se abre su fichero, consultando el i-nodo obtenemos el puntero indirecto doble, y a partir de ahí:

- 1) El primer acceso a disco lo necesitamos para acceder al *cluster* que alberga los punteros indirectos simples.
- 2) Uno de estos punteros indirectos simples corresponde al acceso en curso, y nos lleva de nuevo al disco, para obtener el *cluster* que alberga los punteros a los clusters de datos.
- 3) El tercer acceso a disco se necesita para acceder a los datos de dicho *cluster*.

En total, se necesitan tres accesos a disco. El siguiente diagrama muestra la secuencia de accesos narrada:



9. Un disco duro de 16 Terabytes está formateado con una única partición de i-nodos en Linux que utiliza *clusters* de 4 Kbytes.

- a) ¿Cuántos bits serían necesarios para que los punteros a estos *clusters* puedan direccionar todo el espacio de datos del disco?

$16 \text{ Terabytes/disco} / 4 \text{ Kbytes/cluster} = 2^{32} \text{ clusters/disco}$. Por lo tanto, son necesarios exactamente 32 bits para los punteros a los *clusters*.

- b) Si la partición anterior alberga un fichero `micosas.txt` de 30 Kbytes, ¿cuántos punteros o índices a *clusters* contiene el i-nodo de ese fichero? (considera que el i-nodo no guarda bits de control ni de partición en el puntero, sino únicamente los bits necesarios para direccionar a los *clusters*)

Son necesarios 8 clusters de 4 Kbytes (32 Kbytes), y por lo tanto, es suficiente con 8 punteros directos.

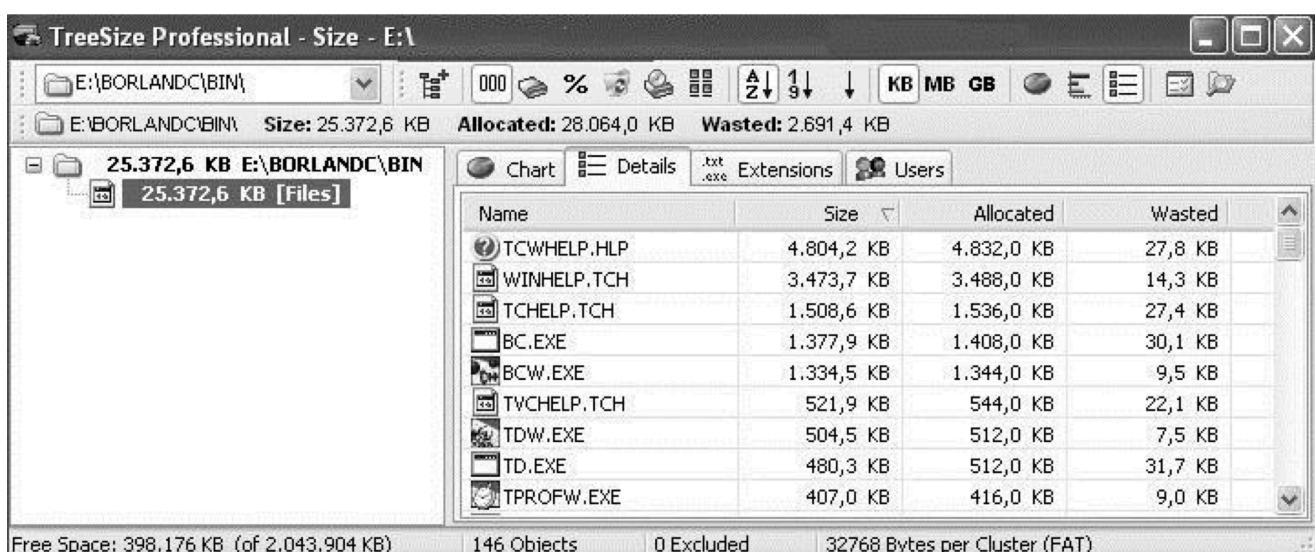
- c) ¿Y si `micosas.txt` ocupara 30 Mbytes?

El puntero indirecto simple accede a un *cluster* que alberga 1024 punteros de 32 bits que pueden alojar hasta 1024 *clusters* de datos. $1024 \text{ clusters} \times 4 \text{ Kbytes/cluster}$ son 4 Mbytes. Por lo tanto, se nos queda corto el puntero indirecto simple y hace falta el puntero indirecto doble.

- d) ¿Y si `micosas.txt` ocupara 30 Gbytes?

El puntero indirecto simple accede a un *cluster* que alberga 1024 punteros de 32 bits que pueden alojar hasta 1024 *clusters* de datos en el disco. $1024 \text{ clusters} \times 4 \text{ Kbytes/cluster}$ suponen 4 Mbytes. El puntero indirecto doble accede a un cluster que contiene 1024 punteros indirectos simples, con capacidad para direccionar $1024 \times 4 \text{ Mbytes}$, es decir, 4 Gbytes. Por lo tanto, hace falta el puntero indirecto triple para llegar a los 30 Gbytes.

10. Un PC bajo Windows XP formatea un disco duro de 2 GBytes bajo un sistema de archivos FAT con tamaño de cluster de 32 Kbytes. A la vista de la siguiente ventana informativa:



Se pide contestar a las siguientes cuestiones:

- a) La columna *Size* que informa del tamaño del archivo no coincide con la columna *Allocated* que indica el espacio que el Sistema Operativo reserva para él. ¿Por qué? ¿Cómo se denomina técnicamente a esta pérdida?

Los archivos sólo pueden reservar espacio por clusters completos, por lo que el remanente que quede del último cluster alojado se desperdicia por la fragmentación interna del disco.

- b) Dicha pérdida se cuantifica en la columna *Wasted* para cada fichero. ¿Cómo podríamos reducir esta pérdida? ¿Provocaría este cambio algún efecto negativo?

Como se observa en la columna *Wasted*, la pérdida para cada fichero oscila entre 0 bytes y el tamaño del cluster, y por tanto puede minimizarse reduciendo el tamaño del cluster. Esto tiene dos efectos negativos: Primero, se aprovecha menos el ancho de banda, cobrando un mayor protagonismo la latencia del disco. Segundo, la FAT ocupa más espacio y cada fichero necesitará apropiarse de un mayor número de clusters, lo que también supone asumir un mayor riesgo de que los clusters no se encuentren alojados consecutivamente en disco y haya que posicionar varias veces los cabezales del brazo de lectura/escritura.

- c) Establecer una fórmula matemática que relacione las variables *Size*, *Allocated* y *Waster*.

$$\text{Allocated} = \text{ceil}(\text{size}/32\text{KB}) \times 32 \text{ KB} \quad \text{Wasted} = \text{Allocated} - \text{Size}$$

- d) ¿Qué tipo de FAT sería suficiente para este disco, FAT12, FAT16 o FAT32?

El disco de 2 Gbytes (2^{31} bytes) se descompone en 2^{16} clusters de 2^{15} bytes/cluster. Por lo tanto, con punteros de 16 bits a los clusters tendríamos suficiente para direccionar todo el espacio del disco, y podría tratarse de una FAT16.

- e) Calcula el espacio de disco ocupado por dicha FAT.

Esta FAT16 ocuparía una tabla de 2^{16} entradas de 16 bits/entrada, para un tamaño total de 128 Kbytes.

- f) ¿Existe algún indicio en la ventana informativa anterior que indique que pueda tratarse de una FAT12, FAT16 o una FAT32?

No.

- g) Si suponemos despreciable el tamaño ocupado por la FAT y el resto de metadatos de la partición del disco, ¿cuál sería el máximo número de ficheros que podría albergar esta partición?

Para maximizar el número de ficheros, tendríamos que hacerlos todos tan pequeños como un cluster, y entonces podríamos alojar tantos ficheros como clusters tiene la partición. Habría entonces $\frac{2 \text{ GB}}{32 \text{ KB}} = \frac{2^{31} \text{ bytes}}{2^{15} \text{ bytes}} = 2^{16}$ ficheros como máximo.

- h) ¿Cuáles son las principales debilidades de un sistema de ficheros basado en FAT como éste respecto a otro basado en índices?

Los inconvenientes que presenta la FAT son básicamente tres. Listados de mayor a menor importancia, tenemos:

- 1) Las cadenas de enlaces compactadas en una estructura de datos común para todos los ficheros supone que si esta estructura sufre alguna anomalía o error, podría arruinar el acceso a un elevado volumen de datos.
- 2) Para acceder al i -ésimo cluster, hay que recorrer toda la cadena de enlaces en la FAT desde el principio, leyendo un total de i entradas de la FAT, que es un proceso lento a no ser que alojemos la FAT en memoria caché.
- 3) Tendríamos ocupados los 128 Kbytes que ocupa la FAT de manera fija, esto es, a pesar de que la partición pueda estar casi completamente vacía de ficheros por parte del usuario.