Sistemas Operativos Avanzados
Práctica 2: Memoria virtual y algoritmos de reemplazo

UAH, Departamento de Automática, ATC-SOL
http://atc1.aut.uah.es

Tema 2

Resumen

El objetivo de esta práctica es estudiar los algoritmos de reemplazo de páginas que se emplean en los mecanismos de gestión de memoria virtual, distinguiendo claramente las tareas que desempeñan el hardware y el sistema operativo en este contexto.

1. Introducción

El objetivo de esta práctica es el estudio de los mecanismos de gestión de memoria virtual. Para ello se simulará el funcionamiento de una MMU (Memory Management Unit) y de la parte correspondiente del sistema operativo.

1.1. El programa `gen_traza`

El programa `gen_traza` se ha creado para alimentar al simulador con una traza de operaciones de lectura/escritura realista. Con ese propósito, `gen_traza` ejecuta un algoritmo de ordenación sobre los datos de un array, y arroja como salida un registro de las operaciones que realiza dicho algoritmo. A continuación se muestra un ejemplo de esta salida:

```
user@host:$ ./gen_traza MER ALE 4
 T8
 L0 E4 L1 E5 L2 E6 L3 E7
 L0 L1 C E4 E5 L2 L3 C
 E6 E7 L4 L6 C E0 L7 C
 E1 L5 C E2 E3 Ordenado ;-)
user@host:$
```

La letra L indica una operación de lectura, y la letra E indica una operación de escritura. En ambos casos, el número que las sigue indica la posición del array a la que accede. La letra C indica una operación de comparación. La letra T aparece sólo una vez, e indica el tamaño total del array. En el ejemplo, el tamaño del array es de 8 elementos, y las operaciones de lectura/escritura se refieren a las posiciones comprendidas entre 0 y 7. Después de ejecutar el algoritmo de ordenación, `gen_traza` recorre el array para comprobar si ha quedado ordenado,

y muestra un mensaje. La traza se puede considerar terminada cuando se llega a la letra O (ordenado) o a la letra D (desordenado).

Cabe destacar que en el ejemplo anterior ha sido necesario acceder a 8 elementos para ordenar solamente 4. Esto se debe a que se ha empleado el algoritmo *mergesort* (ordenación por mezcla de listas ordenadas), que necesita espacio adicional.

El programa `gen_traza` acepta tres parámetros:

1. Algoritmo de ordenación: BUB, INS, SEL, HEA, COM, MER, QUI, o QPA; que indican, respectivamente: burbuja, inserción, selección, montículo (*heapsort*), peine (*combsort*), mezcla (*mergesort*), rápido (*quicksort*), y rápido con pivote aleatorio.
2. Estado inicial del array: ASC, DES o ALE; que indican respectivamente: orden ascendente, orden descendente y orden (o más bien desorden) aleatorio.
3. Número de elementos del array a ordenar (sin contar el espacio adicional requerido por el algoritmo *mergesort*).

1.2. Tamaño de las trazas

La longitud de las trazas generadas por `gen_traza` dependerá del algoritmo elegido, del estado inicial, y del tamaño del array a ordenar.

A continuación se muestra el número de operaciones que necesita realizar cada algoritmo con los tres distintos estados iniciales, y con arrays de 10, 100 y 1000 elementos:

```
Estado inicial: ASC
===================
Tamaño      BUB      INS      SEL      HEA      COM      MER      QUI      QPA

    10       19       19       99      160       46      103      108       85
   100      199      199     9999     2972     2596     1860    10098     1661
  1000     1999     1999   999999    43496    47383    26884  1.0e+06    23792


Estado inicial: DES
===================
Tamaño      BUB      INS      SEL      HEA      COM      MER      QUI      QPA

    10      189      145      109      172       73      107      115       88
   100    19899    14950    10099     3111     3149     1900    10150     1819
  1000  2.0e+06  1.5e+06  1.0e+06    44879    52480    26996  1.0e+06    26229


Estado inicial: ALE
===================
Tamaño      BUB      INS      SEL      HEA      COM      MER      QUI      QPA

    10      133       84      117      165       79      109       71       78
   100    14950     7763    10197     2998     3376     2080     1761     1675
  1000  1.5e+06   741726  1.0e+06    42956    59179    30674    26701    34214
```

Note que hay diferencias muy llamativas, tanto entre unos algoritmos y otros, como entre distintos estados iniciales para un mismo algoritmo. El algoritmo de selección (SEL), por ejemplo, es especialmente lento en todos los casos, mientras que el algoritmo *heapsort* (HEA) es razonablemente rápido en todos los casos. Por otro lado, el algoritmo *quicksort* (QUI) es el más rápido cuando los datos están inicialmente en orden aleatorio, pero es muy lento cuando inicialmente están ya ordenados. Esto se debe a que la implementación de *quicksort* en `gen_traza` siempre elige el primer elemento como pivote. El algoritmo *quicksort* con pivote aleatorio (QPA) es muy rápido en estos experimentos, pero si se conoce la secuencia de números aleatorios que emplea para elegir el pivote, se puede generar un estado inicial que le hace comportarse igual de mal que el *quicksort* normal.

Para más información sobre algoritmos de ordenación, consulte la Wikipedia.

2. Conjuntos de trabajo

El conjunto de trabajo de una sección de programa es el grupo de páginas de memoria que referencia. A lo largo de la ejecución de un proceso, hay periodos en los que se centra en un conjunto de trabajo pequeño, reutilizando durante mucho tiempo unas pocas páginas, y hay otros periodos en los que referencia rápidamente muchas páginas diferentes.

Cuanto menor sea el conjunto de trabajo, más probabilidades habrá de que las páginas referenciadas estén presentes en marcos de memoria física. Por otro lado, cuanto más amplio sea el conjunto de trabajo, más probabilidades habrá de que algunas páginas sean desalojadas de sus marcos para hacer sitio a otras, lo que hará que después se produzcan fallos de página al referenciarlas de nuevo.

El programa `calcular_cdt` hace un cálculo rudimentario del conjunto de trabajo a lo largo de una ejecución de `gen_traza`. La figura 1 muestra una gráfica generada a partir de la salida de `calcular_cdt` para una ejecución de `gen_traza` con el algoritmo *mergesort*. El pico inicial se debe a que esta implementación de *mergesort* empieza copiando todo el array a ordenar en el array temporal. A continuación, el conjunto de trabajo se reduce considerablemente por el orden en que se van construyendo listas ordenadas a partir de listas ordenadas más pequeñas: primero se forma una pareja ordenada con los elementos 0 y 1, luego se forma otra pareja ordenada con los elementos 2 y 3, y entonces se unen esas dos parejas en una lista de 4 elementos. Entonces se forma otra lista de 4 elementos siguiendo el mismo procedimiento, y se mezcla con la anterior, para generar la primera lista de 8 elementos, y así sucesivamente.

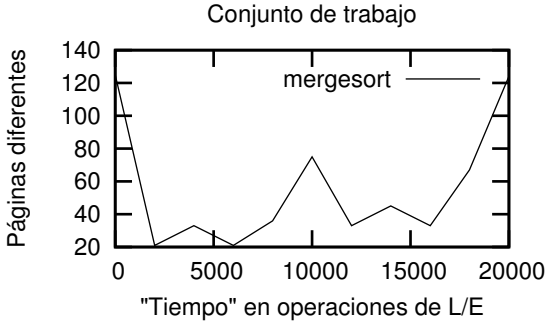

Figura 1: Gráfica del conjunto de trabajo de una ejecución del algoritmo *mergesort*

Según se van agrandando las listas ordenadas, se accede más rápidamente a más posiciones diferentes de la memoria, con lo que aumenta el conjunto de trabajo. La última mezcla de listas ordenadas accede en un corto espacio de tiempo a todas las posiciones del array, haciendo que el conjunto de trabajo se agrande tanto como en el pico inicial.

El programa `calcular_cdt` acepta 5 parámetros:

1. Número de elementos que caben en una página.
2. Número de operaciones por intervalo. Se contará el número de páginas diferentes referenciadas a lo largo de cada intervalo.
3. Algoritmo (como en `gen_traza`).
4. Estado inicial (como en `gen_traza`).
5. Número de elementos del array a ordenar (como en `gen_traza`).

La salida de `calcular_cdt` tiene el formato idóneo para alimentar al programa `gnuplot`. Genere la gráfica del conjunto de trabajo de otro algoritmo de ordenación como en el siguiente ejemplo:

```
user@host:$ ./calcular_cdt 16 2000 QPA ALE 1000 >cdt_qpa.txt
user@host:$ gnuplot

        G N U P L O T
        Version 4.2 patchlevel 5
        [..]

Terminal type set to 'wxt'
gnuplot> set encoding iso_8859_1
gnuplot> set xlabel "\"Tiempo\" en operaciones de L/E"
gnuplot> set ylabel "P\341ginas diferentes"
gnuplot> set title "Conjunto de trabajo"
gnuplot> plot "cdt_qpa.txt" using 1:3 title "QPA" with lines
gnuplot> exit
user@host:$
```

Una vez introducida la orden `plot` debería aparecer la gráfica en una nueva ventana. Las órdenes para `gnuplot` se pueden almacenar en un archivo de texto, de forma que no haya que teclearlas cada vez que se vaya a pintar la gráfica. En tal caso, hay que ejecutar `gnuplot` pasándole como parámetro el nombre del archivo que contiene las órdenes. Además conviene añadir al final del archivo la orden `pause -1 "Pulse ENTER"` para evitar que la ventana de la gráfica se cierre inmediatamente.

El programa `gnuplot` también puede grabar directamente la gráfica en un archivo EPS (*Encapsulated PostScript*). Sin ir más lejos, la gráfica de la figura 1 ha sido generada mediante un archivo de texto con las siguientes órdenes:

```
set terminal postscript portrait enhanced \
    mono dashed lw 1 "Arial" 9

set encoding iso_8859_1
set out "grafica_cdt.eps"

set size 0.50, 0.18
set size ratio 0.5

set xlabel "\"Tiempo\" en operaciones de L/E"
set ylabel "P\341ginas diferentes"
set title "Conjunto de trabajo"

plot "tabla_cdt.txt" using 1:3 title "mergesort" with lines
```

Compare la evolución del conjunto de trabajo del algoritmo de la burbuja con la del algoritmo de inserción. Utilice un array de 1000 elementos desordenados (en orden aleatorio), páginas de 16 elementos e intervalos de 100000 operaciones.

3. Simulador de gestión de memoria virtual

El resto de esta práctica consistirá en completar, y después modificar, un programa que simula el funcionamiento de una MMU (*Memory Management Unit*) y de la parte del Sistema Operativo que gestiona la memoria virtual. El simulador está programado casi íntegramente, y sólo falta añadir algunas funciones para poder ejecutarlo.

Al igual que en `calcular_cdt`, la función `main` del simulador ejecuta a `gen_traza` e interpreta su salida estándar. Por cada operación de lectura/escritura, invoca a la función `sim_mmu`, que simula el acceso a la dirección virtual especificada.

Abra el archivo `sim_paginacion.h` y lea detenidamente la declaración del tipo de estructura `spagina`. Las estructuras de este tipo representan entradas de la tabla de páginas. Lógicamente, la tabla de páginas se simula mediante un array de estructuras `spagina`. El campo `presente` indica si la página se encuentra cargada en memoria física (ocupando un marco). Si este campo vale 0, el resto de campos se consideran inválidos. Si vale 1, entonces el campo `marco` almacena el número del marco físico en el que está cargada la página, y el campo `modificada` indica si se ha escrito en la página desde que ésta fue cargada, o si sólo se ha leído. Los campos `referenciada` y `timestamp` servirán para simular sistemas con reemplazo FIFO (*First In → First Out*) con 2ª oportunidad y LRU (*Least Recently Used*) respectivamente.

Los campos `presente`, `modificada` y `referenciada` se almacenan ocupando un byte cada uno para que el código del simulador sea más legible. En una simulación más fiel a la realidad, habría que empaquetarlos de forma que ocupasen sólo un bit cada uno.

La tabla de páginas debe ser conocida y manipulada tanto por el hardware como por el sistema operativo. En el simulador, las funciones `sim_mmu` y `referenciar_pagina` (implementadas en

`sim_pag_aleatorio.c`) son las que hacen el trabajo del hardware, mientras que todas las demás simulan el comportamiento del sistema operativo.

Observe la función `referenciar_pagina`. Al igual que todas las funciones que trataremos aquí, recibe un puntero `S` que apunta a una estructura que almacena el estado del sistema simulado. Entre otras cosas, esa estructura contiene un puntero a la tabla de páginas, y unos contadores de referencias a memoria. Además, la función recibe el número de la página a la que se está accediendo, y el tipo de operación (L/E). La función `referenciar_pagina` incrementa el contador de lecturas o el de escrituras (según el tipo de operación) y además, si la operación es de escritura, accede a la tabla de páginas para activar el bit `modificada` de la entrada correspondiente a la página en cuestión.

Complete `sim_mmu` siguiendo las instrucciones que se indican a continuación. En primer lugar, `sim_mmu` debe calcular el número de página y el desplazamiento a partir de la dirección virtual. El tamaño de página está almacenado en el campo `tampag`[1] de la estructura apuntada por `S`.

```
pagina         = dir_virtual / S->tampag;  // Cociente
desplazamiento = dir_virtual % S->tampag;  // Resto
```

En una simulación más realista, el tamaño de página se fijaría de forma que coincidiera con una potencia de 2. Gracias a ello, el cálculo anterior consistiría simplemente en partir en dos trozos la dirección virtual (almacenada en binario).

A continuación, `sim_mmu` debe comprobar que el acceso a la dirección especificada es legal:

```
if (pagina<0 || pagina>=S->numpags)
{
    S->numrefsilegales ++;
    return ~0U;              // Devolver dir. física FF...F
}
```

Una vez que se ha calculado la dirección y se ha comprobado que es legal, hay que consultar la tabla de páginas para ver si la página en cuestión se encuentra cargada en memoria física. Si no lo está, hay que provocar un *trap* de fallo de página, es decir, interrumpir el proceso e invocar al sistema operativo para que resuelva el problema. En el simulador, la función `tratar_fallo_de_pagina` hará el papel de esa rutina del sistema operativo, ocupándose de cargar la página en algún marco de memoria.

```
if (!S->tdp[pagina].presente)                  // No presente:
    tratar_fallo_de_pagina (S, dir_virtual);   // FALLO DE PÁG.
```

Una vez que el sistema operativo ha cargado la página en un marco y ha modificado la tabla de páginas consecuentemente, se reanuda operación de acceso a memoria.

[1] En la simulación, cada posición de memoria contiene un elemento del array a ordenar. El tamaño de página se especifica en número de elementos. El número de bits o bytes que ocupa un elemento es irrelevante en esta práctica.

El siguiente paso es traducir la dirección virtual a dirección física:

```
// Ahora ya está presente

marco = S->tdp[pagina].marco;          // Calcular dirección
dir_fisica = marco * S->tampag +       // física
             desplazamiento;
```

De nuevo, si el tamaño de página fuese una potencia de 2, la multiplicación y la suma se reducirían simplemente a la operación de concatenar dos números binarios.

También hay que marcar la página como refrenciada:

```
referenciar_pagina (S, pagina, op);
```

Ya sólo falta volcar la información del acceso a memoria por pantalla si el usuario ordenó ejecutar el simulador en modo D (detallado):

```
if (S->detallado)
    printf ("\t%c %u==P%d(M%d)+%d\n",
            op, dir_virtual, pagina, marco, desplazamiento);
```

A continuación estudiaremos el papel del sistema operativo en la gestión de la memoria virtual. A ese respecto, durante la ejecución del proceso, el punto de entrada al sistema operativo es la rutina de tratamiento del fallo de página. Pero antes de abordarla, estudiaremos otras estructuras de datos que necesita manejar el sistema operativo.

Si sólo dispusiera de la tabla de páginas, el sistema operativo tendría que hacer costosas búsquedas secuenciales por toda la tabla para llevar a cabo las siguientes operaciones:

- Determinar si un marco está libre u ocupado
- Averiguar qué página está almacenada en un marco determinado
- Encontrar un marco libre (suponiendo que lo haya)
- Elegir un marco para reemplazar la página que lo ocupa (en caso de que no haya ningún marco libre)

El sistema operativo solventa este problema manteniendo una tabla de marcos. La MMU no necesita conocer la existencia de la tabla de marcos porque ésta es mantenida exclusivamente por el sistema operativo.

Observe la declaración del tipo de estructura `smarco` en `sim_paginacion.h`. El campo `pagina` indica el número de la página que está almacenada en el marco. El campo `sig` sirve para mantener los marcos libres organizados en una lista enlazada. Almacena el número del siguiente marco en la lista.

La lista enlazada de marcos libres es circular. El campo `listalibres` de la estructura de tipo `ssistema` (a la que apunta `S`) almacena el número del último marco de la lista. Para llegar al primero de la lista sólo hay que consultar el campo `sig` del último, porque la lista es circular. La función `iniciar_tablas` se ocupa de que, al principio, la lista de marcos libres contenga todos los marcos. Cuando la lista esté vacía, `S->listalibres` valdrá -1.

Complete la función `tratar_fallo_de_pagina` siguiendo las instrucciones que se indican a continuación. En primer lugar hay que calcular el número de página que provocó el fallo. De paso, incrementamos el contador de fallos de página y mostramos un mensaje por pantalla si el simulador está en modo D (detallado).

```
S->numfallospag ++;
pagina = dir_virtual / S->tampag;

if (S->detallado)
    printf ("@ ¡FALLO DE PÁGINA en P%d!\n", pagina);
```

En caso de que haya marcos libres, bastará sacar uno de la lista (el primero es el que está más a mano) y ocuparlo con la página solicitada:

```
if (S->listalibres!=-1)     // Si hay marcos libres:
{
    ult = S->listalibres;           // Último de la lista
    marco = S->tdm[ult].sig;        // Tomar el sig. (el 1º)

    if (marco==ult)                 // Si son el mismo, es que
        S->listalibres = -1;        // sólo quedaba uno libre
    else
        S->tdm[ult].sig = S->tdm[marco].sig; // Si no, puentear

    ocupar_marco_libre (S, marco, pagina);
}
```

En caso de que no haya marcos libres, habrá que elegir uno de los que están ocupados y desalojarlo para poder cargar en él la página solicitada. La política de reemplazo (el algoritmo que elige la página víctima) y la operación de reemplazo propiamente dicha están implementadas por separado.

```
else                            // Si _no_ hay marcos libres:
{
    victima = elegir_pagina_para_reemplazo (S);

    reemplazar_pagina (S, victima, pagina);
}
```

Observe el código de la función `reemplazar_pagina`. Cabe destacar que, en un sistema operativo real, esta rutina no sólo se ocupa de actualizar las tablas y cargar la nueva página en el marco, sino que también tiene que volcar la página víctima a disco en caso de que haya sido modificada mientras estaba cargada.

Programe la función `ocupar_marco_libre`. Sólo debe hacer el enlace página-marco, y marcar adecuadamente los bits de la página. En un sistema real, esta función también leería la página del disco para ponerla en el marco.

3.1. Reemplazo aleatorio

Edite el archivo `Makefile` y añada al objetivo `all` el programa `sim_pag_aleatorio`. Compile y ejecute el simulador:

```
user@host:$ make

user@host:$ ./sim_pag_aleatorio 1 3 HEA DES 4 D
```

La orden anterior especifica un tamaño de página de un solo elemento, un tamaño de memoria física de tres páginas, modo D (detallado) y ejecución de `gen_traza` con parámetros `HEA DES 4` (algoritmo *heapsort*, estado inicial de orden descendente, y array a ordenar de cuatro elementos). En la instalación actual del laboratorio, el informe resultante debería coincidir con el siguiente:

```
---------- INFORME GENERAL ----------

Referencias de lectura:   20
Referencias de escritura: 17
Fallos de página:         7
Páginas volcadas a disco: 2

---------- TABLA DE PÁGINAS ---------

   PÁGINA   Presente      Marco   Modificada
       0        1            1        1
       1        1            0        1
       2        1            2        1
       3        0            -        -

---------- TABLA DE MARCOS ----------

     MARCO    Página   Presente   Modificada
       0          1        1          1
       1          0        1          1
       2          2        1          1

--------- INFORME REEMPLAZO ---------

Reemplazo aleatorio (no hay info. específica)

-------------------------------------

FALLOS DE PÁGINA: --->> 7 <<---
```

3.2. Reemplazo LRU

A continuación haremos una versión del simulador con otra política de reemplazo. Haga una copia del archivo `sim_pag_aleatorio.c` y llame al nuevo archivo `sim_pag_lru.c`. Edite el archivo `Makefile` y añada al objetivo `all` el programa `sim_pag_lru`. Modifique el comentario de la cabecera de `sim_pag_lru.c` para que se corresponda con el nombre del archivo.

La política de reemplazo LRU (*Least Recently Used*) consiste en elegir como víctima del reemplazo a la página utilizada menos recientemente con la esperanza de que no sea referenciada tampoco en el futuro cercano. Implemente esta política mediante las siguientes modificaciones:

1. Añada instrucciones a la función `referenciar_pagina` para que guarde el valor del reloj en el campo `timestamp` de la página accedida, y después incremente el valor del reloj. Añada también una comprobación para imprimir un mensaje de advertencia en caso de que el reloj se desborde y vuelva a valer 0.
2. En `elegir_pagina_para_reemplazo`, implemente una búsqueda secuencial del marco ocupado cuya página tenga el menor timestamp. Cambie "al azar" por "LRU" en el mensaje de la llamada a `printf`. Borre la función `aleatorio`.
3. En `mostrar_tabla_de_paginas`, añada una columna que muestre el valor reloj−*timestamp* de las páginas presentes en memoria.
4. Haga que `mostrar_informe_reemplazo` muestre el valor del reloj y los *timestamp* mínimo y máximo de las páginas presentes en memoria.

Compile y ejecute el nuevo programa `sim_pag_lru`. Verifique que los resultados tienen sentido. Puede usar los números de fallos de página de la columna LRU en el cuadro 1 como referencia.

Cuadro 1: Fallos de página según algoritmo de reemplazo

Parámetros	Aleatorio	LRU	FIFO	FIFO2ª	Óptimo
16 3 HEA DES 100	280	282	283	285	171
16 8 HEA DES 1000	3101	2436	2877	2642	1360
16 32 HEA DES 10000	15614	10802	13427	11211	8792
16 3 MER DES 100	158	104	119	118	83
16 8 MER DES 1000	1111	905	907	898	722
16 32 MER DES 10000	10190	9549	9458	9530	8146

3.3. Reemplazo FIFO

Haga una nueva copia de `sim_pag_aleatorio.c` y llámela `sim_pag_fifo.c`. Repita los pasos iniciales del apartado anterior, esta vez para crear el programa `sim_pag_fifo`.

Implemente la política de reemplazo FIFO. Esta política consiste en desalojar las páginas en el mismo orden en el que fueron cargadas. Es decir, la página que entra primero, sale primero (*First In* → *First Out*). Para ello, mantenga una lista circular enlazada de los marcos ocupados como la que se muestra en la figura 2. El campo `listaocupados` de la estructura `ssistema` apuntará al último elemento.

Haga las siguientes modificaciones:

1. Como en el apartado anterior, elimine los elementos que correspondan a la política de reemplazo aleatorio.
2. En `ocupar_marco_libre`, añada el marco al final de la lista de marcos ocupados (añádalo entre el último y el primero, y después haga que `listaocupados` apunte al elemento añadido). Tenga en cuenta que, inicialmente, la lista está vacía y `listaocupados` vale -1.
3. En `elegir_pagina_para_reemplazo`, escoja como víctima a la página del primer marco de la lista (el que sigue al último), y después muévalo al final (haga que `listaocupados` apunte al marco elegido).
4. En `mostrar_informe_reemplazo`, muestre la lista de marcos ocupados.

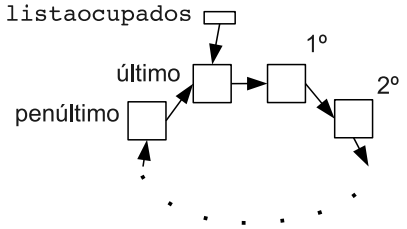

Figura 2: Lista circular de marcos ocupados

Compile el programa y ejecútelo en modo D (detallado) con un número reducido de páginas y marcos para verificar que el reemplazo se hace en orden FIFO. Después ejecútelo con los parámetros de alguno de los ejemplos del cuadro 1 para comprobar que el número de fallos de página coincide.

3.4. Reemplazo FIFO con segunda oportunidad

Haga una nueva copia a partir de `sim_pag_fifo.c` y llámela `sim_pag_fifo2op.c`. Repita los pasos iniciales de los apartados anteriores, esta vez para crear el programa `sim_pag_fifo2op`.

Modifique `sim_pag_fifo2op.c` para que implemente la política de reemplazo FIFO con 2ª oportunidad. Esta política consiste en dar una segunda oportunidad al primer marco de la cola FIFO, pero sólo si la página ha sido referenciada desde la última vez que el marco estuvo el primero en la cola. La segunda oportunidad consiste en indultar a la página de ese marco, moviendo el marco al final de la lista de ocupados, pero poniendo a cero el bit de referencia de la página.

Realice los siguientes cambios:

1. En `referenciar_pagina`, ponga a 1 el bit de referencia de la página.
2. En `elegir_pagina_para_reemplazo`, programe un bucle que salte los marcos de páginas referenciadas (poniendo a 0 su bit de referencia y haciendo que `S->listaocupados` también avance) hasta encontrar un marco cuya página tenga el bit de referencia a 0 (esa será la víctima del reemplazo).

3. En `mostrar_tabla_de_paginas` y `mostrar_tabla_de_marcos`, añada una columna que muestre el bit de referencia de las páginas.

4. Modifique también `mostrar_informe_reemplazo` para que muestre el bit de referencia de la página almacenada en cada marco.

Compile el programa y ejecútelo en modo D (detallado) con un número reducido de páginas y marcos para verificar que el reemplazo se hace en orden FIFO con 2^a oportunidad. Después ejecútelo con los parámetros de alguno de los ejemplos del cuadro 1 para comprobar que el número de fallos de página coincide.

3.5. Reemplazo óptimo

La mejor política de reemplazo posible sería la que reemplazara, en cada caso, la página que más tiempo fuese a tardar en ser referenciada. Para ello, el sistema operativo necesitaría predecir el futuro eficientemente y con exactitud[2].

Obviamente, el reemplazo óptimo no es factible en un sistema real. No obstante, es interesante poder simularlo porque su comportamiento es el modelo ideal al que un buen algoritmo de reemplazo debería acercarse.

El programa `sim_pag_optimo` simula el reemplazo óptimo haciendo "trampa", es decir, guardando toda la traza en memoria antes de comenzar la simulación para después saber en cada paso lo que va a ocurrir después y decidir en consecuencia.

Ejecute `sim_pag_optimo` en modo D (detallado) con un número reducido de páginas y marcos, y observe el resultado. Por ejemplo:

```
user@host:$ ./sim_pag_optimo 1 1 BUB ALE 2 D | grep @
@ ¡FALLO DE PÁGINA en P0!
@ Alojando P0 en M0
@ ¡FALLO DE PÁGINA en P1!
@ Eligiendo P0 (tardará 1 operaciones en ser utilizada de nuevo) de M0
                                                    para reemplazarla
@ Reemplazando víctima P0 por P1 en M0
@ ¡FALLO DE PÁGINA en P0!
@ Eligiendo P1 (tardará 1 operaciones en ser utilizada de nuevo) de M0
                                                    para reemplazarla
@ Reemplazando víctima P1 por P0 en M0
@ ¡FALLO DE PÁGINA en P1!
@ Eligiendo P0 (no se usará más) de M0 para reemplazarla
@ Volcando P0 modificada a disco para reemplazarla
@ Reemplazando víctima P0 por P1 en M0
```

[2]La manera más eficiente de predecir el futuro con exactitud es esperar hasta que éste ocurra. Por ahora. En el futuro ya veremos...