Ejercicios de

GESTION DE MEMORIA

Ejercicio 1

Sea un computador que dispone de 36 MB de memoria principal y cuyo sistema operativo ocupa 4 MB sin incluir las estructuras necesarias para el gestor de memoria. En este computador se prevé la ejecución de programas con un espacio de direcciones lógico compuesto por tres segmentos: texto, datos y pila, siendo los tamaños ***medios*** *de estos segmentos: 264 KB, 124 KB y 124 KB respectivamente.*

Se desea implementar un gestor de memoria, siendo los posibles esquemas de gestión a implementar (suponiendo que el hardware puede soportar todos ellos) los siguientes:

1. *Particiones variables, tal que cada proceso ocupa una única partición*
2. *Paginación simple*
3. *Segmentación simple*
4. *Memoria virtual paginada*
5. *Memoria virtual con segmentación paginada*

Para los sistemas basados en páginas, considere que el tamaño de la página es de 4 Kbytes y que cada entrada de la tabla de páginas ocupa 32 bits. En el caso de esquemas con segmentación, el tamaño máximo del segmento es de 16 Mbytes. Por último, suponga que en los modelos con memoria virtual se usa una política de asignación fija que otorga a cada proceso un número fijo de 16 marcos de página.

Se pide analizar para cada uno de estos esquemas los siguientes aspectos:

a.- La memoria utilizada por el sistema de gestión de memoria para las estructuras de datos necesarias.

b.- La memoria perdida debido al esquema de gestión de memoria utilizado, indicando el motivo de dicha pérdida.

c.- El grado de multiprogramación que se puede alcanzar en el sistema con los tamaños medios propuestos para los procesos.

d.- El tamaño máximo del proceso que se puede ejecutar en cada uno de los distintos esquemas de gestión de memoria.

Observaciones: *En los casos en que no se disponga de otros datos, y sea aplicable, considérese que se pierde un 25 % de la memoria debido al esquema de gestión utilizado.*

Solución

Dado el espacio ocupado por el SO, quedan 32 MB para la ejecución de los programas y para las estructuras de datos del gestor de memoria.

1. Particiones variables

En este esquema de gestión de memoria, las particiones son variables tanto en número como en tamaño. Cuando un proceso se trae a memoria se le asigna exactamente la memoria que necesita y no más.

a) En este caso basta con un par de palabras por proceso que indiquen la dirección de memoria donde comienza el proceso (registro base) y la longitud del mismo (registro límite). Con este par de palabras se resuelve tanto el problema de la reubicación como el de la protección.

Aparte de estas dos palabras, el sistema de gestión de memoria necesita conocer en todo momento, qué partes de la memoria están disponibles (huecos). Para ello requerirá alguna estructura como puede ser una lista enlazada.

b) En este esquema de gestión de memoria se produce fragmentación externa. Ésta se presenta cuando en memoria existe suficiente espacio para cargar un nuevo proceso, pero sin embargo éste no es contiguo; la memoria está compuesta por bloques que son demasiados pequeños para contener un proceso.

Según el enunciado, la memoria que se pierde como consecuencia de la fragmentación externa es un 25 % de la memoria disponible, es decir, 8 MB.

c) El tamaño medio de un proceso en este computador es de 512 KB. Dado que la cantidad de memoria que en un momento determinado se pierde como consecuencia de la fragmentación externa es de 8 MB, los procesos ocupan en el sistema 24 MB. Por lo tanto el grado de multiprogramación será:

n = (24·1024) / 512 = 48 procesos.

d) El tamaño máximo del proceso que se puede ejecutar en este computador viene determinado por el tamaño de la memoria física disponible para la ejecución de programas de aplicación, que en este caso es de 32 MB, menos el espacio ocupado por las estructuras consideradas en el apartado a.

2. Paginación simple:

Utilizando paginación simple tanto el proceso como la memoria se dividen en un conjunto de particiones fijas y de igual tamaño, denominadas páginas y marcos de página respectivamente. Cuando se introduce un proceso en el sistema, se cargan todas sus páginas en la memoria.

a) En este caso se necesita un tabla de páginas por proceso que permita realizar la traducción de direcciones lógicas a físicas, indicando la dirección de memoria física donde se encuentra el marco correspondiente a cada página del proceso. Asimismo se necesita una lista de marcos libres.

b y **c)** El número medio de páginas que ocupa cada proceso es 512 / 4 = 128 páginas. El tamaño mínimo que ocupa la tabla de páginas de cada proceso en memoria es 128·4 = 512 bytes (teniendo en cuenta que cada entrada de la tabla de páginas ocupa 4 bytes).

La suma de todas las tablas de páginas más la de marcos libres es de 8192 (el número total de marcos disponibles), luego el espacio total requerido es de 8192·4 KB = 32768 KB = 8 páginas. En realidad el gestor de memoria necesitará más espacio, puesto que las tablas de páginas son elementos dinámicos que se crean y destruyen, lo que hace que su gestión requiera más espacio que el mínimo calculado.

La memoria que se pierde en este caso se debe a la fragmentación interna que se puede producir en la última página de cada proceso, y que en media es la mitad de una página, es decir, 2 KB por proceso. Para calcular la memoria que se pierde como consecuencia este tipo fragmentación, es necesario calcular en primer lugar el grado de multiprogramación, que en este caso es:

n = 8184 / 128 = 63 procesos.

Según esto, la memoria que se pierde en el sistema como consecuencia de la fragmentación interna es 2 KB·63 = 126 KB, lo que representa menos del 1 % de la memoria disponible para la ejecución de programas de usuario.

d) El tamaño máximo del proceso que puede ejecutar en este caso viene determinado por el tamaño de la memoria principal, descontando el espacio necesario para representar la tabla de páginas del proceso, y que en este caso es 8184 páginas.

Por lo tanto el tamaño máximo de un proceso será 8184·4 KB = 32736 KB.

3. Segmentación simple

Empleando segmentación simple, el programa se divide en un conjunto de segmentos, no siendo necesario que todos los segmentos tengan la misma longitud ni estén contiguos en memoria. En este caso cada proceso constará de un segmento de texto, uno de datos y otro de pila. Cuando se trae un proceso a memoria será necesario cargar todos los segmentos del proceso en la misma.

a) El empleo de segmentación resulta muy similar al de particiones variables. Será necesario una pequeña tabla de segmentos, bastando en este caso con un conjunto de registros base y límite para cada uno de los tres segmentos, es decir, 6 palabras por proceso.

Al igual que en el caso de particiones variables, el sistema de gestión de memoria necesitará algún tipo de estructura para conocer la ocupación de la memoria en todo momento, siendo aplicable lo comentado en el apartado 1.

b y **c)** En este esquema de gestión de memoria se produce al igual que en el apartado 1 fragmentación externa, que en este caso vuelve a ser de 8 MB (el 25 % de la memoria disponible para programas de usuario). De igual forma, el grado de multiprogramación que se puede alcanzar es equivalente al del apartado 1 (48 procesos), con la diferencia de que los procesos no se encuentran contiguos en memoria.

Aunque en ambos casos, la fragmentación externa que se produce es la misma, ya que según el enunciado se pierde el 25 % de la memoria, esto no es lo que ocurre en situaciones reales. Generalmente cuando se reduce el tamaño del segmento también se reduce la memoria que se pierde como consecuencia de la fragmentación externa. Para comprobar esto, vamos a suponer que el porcentaje que se pierde como consecuencia de la dicha fragmentación es un 50 % del tamaño del segmento que se ubica en memoria y que se cumple la regla de cincuenta por ciento según la cual si el número de segmentos en memoria es n el número de huecos es n/2.

En el caso de particiones variables, se produce fragmentación externa cuando los huecos existentes en memoria son de tamaño menor a 512 KB (el tamaño medio de un proceso en este computador). Si el tamaño del hueco es un 50 % del tamaño de segmento a ubicar (512 KB), entonces se cumple que la cantidad de memoria ocupada por los procesos más la que ocupan los huecos es igual a la memoria total, es decir:

$512 \cdot n + n/2 \cdot (0.5 \cdot 512) = 32 \cdot 1024 \rightarrow n = 51$ procesos.

Cuando se emplea segmentación simple, la fragmentación externa viene determinada por el tamaño del segmento más pequeño, que en este caso es de 124 KB. El número de segmentos en este caso es 3n y por lo tanto el número de huecos 3n/2. El grado de multiprogramación por tanto en este caso será:

$512 \cdot n + 3n/2\ (124 \cdot 0.5) = 32 \cdot 1024 \rightarrow n = 54$ procesos.

Lo que nos demuestra que podríamos albergar más procesos en memoria utilizando segmentación simple.

d) El máximo proceso que se puede ejecutar en este computador viene determinado por la memoria física disponible, que es 32 MB, menos el espacio ocupado por las estructuras consideradas en el apartado a.

4. Memoria virtual paginada.

En este caso, a diferencia de un esquema de paginación simple, no es necesario que todas las páginas del proceso se encuentren en memoria principal para su ejecución.

a) Se necesita al igual que en el apartado 2, una tabla de páginas por proceso cuyo tamaño mínimo (ver apartado 2) será de 512 bytes.

b y **c)** La memoria que se pierde en este caso se debe a la fragmentación interna que se produce en la última página del espacio de direcciones virtuales del proceso. Dado que el gestor de memoria otorga a cada proceso un número fijo de 16 marcos de página, la probabilidad de que la última página del proceso se encuentre en memoria principal en un momento determinado es 16/128. Para calcular cantidad de memoria total perdida como consecuencia de esta fragmentación, determinaremos previamente el grado de multiprogramación en el sistema.

El tamaño que ocupa cada proceso en memoria principal viene dado por los 16 marcos de página asignados a cada proceso y por el tamaño de la tabla de páginas cuyo tamaño mínimo es de 512 bytes. Según esto el grado de multiprogramación será:

$n(16 \cdot 4 + 0{,}5) = 32 \cdot 1024 \rightarrow n = 508$ procesos, un 700 % mayor que el obtenido en el apartado 2.

De acuerdo con esto la mínima memoria empleada para las tablas de página será de $508 \cdot 512$ bytes = 254 KB, y la memoria total que se pierde debido a la fragmentación interna será $(16/128) \cdot 508 \cdot 2$ KB = 127 KB, prácticamente igual que empleando paginación simple.

d) El tamaño máximo del proceso que se puede ejecutar utilizando memoria virtual paginada vendrá determinado por el menor de: a) la cantidad de memoria que permita direccionar la arquitectura de este computador; b) el tamaño reservado para el área de *swap* (más la memoria principal en su caso).

5. Memoria virtual con segmentación paginada

En este caso el proceso consta de tres segmentos, cada uno de los cuales a su vez se página para evitar de esta forma el problema de la fragmentación externa que aparecía en el caso de segmentación simple.

a) Se necesita una estructura para representar cada una de las tres regiones del proceso. Esta estructura contendrá para cada región: la dirección virtual de comienzo del segmento, la longitud del mismo y la

dirección de la tabla de páginas correspondiente a ese segmento. Por lo tanto se necesitarán tres palabras por región, es decir, nueve palabras por proceso.

El mínimo tamaño necesario para cada una de estas tablas es el siguiente: 66 entradas para el segmento de texto, es decir, 66·4 = 264 bytes, y 31 entradas, 31·4 = 124 bytes, para los segmentos de datos y pila.

b y **c)** La memoria que se pierde empleando memoria virtual con segmentación paginada se debe a la fragmentación interna que se produce en la última página del espacio de direcciones virtuales correspondiente a cada segmento, es decir, se puede producir fragmentación interna en tres páginas del proceso (en memoria virtual). De forma análoga al apartado anterior, la probabilidad de que alguna de estas páginas sea la que se almacene en los 16 marcos de memoria que se asignan a cada proceso es (3·16) / 128. Para calcular la memoria total perdida como consecuencia de esta fragmentación, calcularemos en primer lugar el grado de multiprogramación.

El número de procesos que puede haber en memoria viene determinado por la cantidad de páginas residentes en memoria, 16 en este caso, y por el tamaño ocupado por las tablas de segmentos y de páginas, cuyo tamaño mínimo en este caso es de 264 + 124 + 124 + (9·4) = 548 bytes. Según esto:

$$n(16\cdot 4 + 548/1024) = 32\cdot 1024 \Rightarrow n = 507 \text{ procesos.}$$

La memoria perdida debido a la fragmentación interna en este caso es

$$(3\cdot 16/128)\cdot 507\cdot 2 \text{ KB} = 380 \text{ KB}.$$

d) El tamaño máximo del proceso que se puede ejecutar en este caso viene determinado por el tamaño máximo de cada uno de los segmentos que es de 16 MB. Por lo tanto, el tamaño máximo será 16·3 = 48 MB.

Ejercicio 2

Sea un sistema de memoria virtual sin preasignación de swap y que usa la técnica del almacenamiento intermedio (buffering) de páginas. En dicho sistema se pretenden estudiar los distintos estados por los que va pasando una página de un proceso durante la ejecución del mismo, dependiendo del segmento al que pertenezca la página. Para ello vamos a considerar como estado de una página en un determinado momento, el lugar de donde la obtendría el proceso al accederla en ese instante. Se distinguen, por lo tanto, los siguientes ***estados****:*

- *Del fichero (**F**).*
- *De la memoria principal asignada al proceso (**P**).*
- *De la lista de páginas modificadas (**M**) (almacenamiento intermedio).*
- *De la lista de páginas libres (**L**) (almacenamiento intermedio).*
- *Del dispositivo de swap (**S**).*

Asimismo, para analizar las transiciones entre estados, se considerarán, al menos, los siguientes ***eventos****:*

- *Fallo de página (**F**).*
- *Extracción (**XT**): se le quita la página al proceso pasando el marco al almacenamiento intermedio.*
- *Expulsión (**XP**): se elimina la página de la memoria principal liberando el marco.*
- *Escritura de la página al dispositivo o fichero (**W**) permaneciendo en el almacenamiento intermedio.*

Por último, las transiciones entre estados, además de estar dirigidas por estos eventos, podrán depender de condiciones tales como que la página haya sido modificada o no.

Se pide dibujar, para los tipos de páginas que se plantean a continuación, el diagrama de estados que muestra todos los estados en los que puede estar una página, indicando con flechas los eventos y/o condiciones que producen las transiciones. En cada caso, se usarán los estados y eventos que sean pertinentes, y se explicarán los aspectos más relevantes de cada diagrama identificando cuando se producen operaciones de acceso al disco.

a) Una página de código.

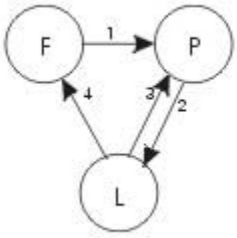
b) Una página de datos con valor inicial. ¿Se podrían servir del fichero ejecutable sucesivos fallos asociados a una misma página de este tipo?

c) Una página de datos sin valor inicial. Una de las situaciones que se debe considerar es el caso de una página de este tipo que se accede por primera vez con una lectura y que se expulsa sin haber realizado más accesos sobre la misma.

d) Una página de un fichero proyectado (mmap).

Solución

a) Las páginas de código no se modifican por lo que nunca pasarán a la lista de páginas modificadas del almacenamiento intermedio. Además, debido a ello, no se almacenarán en el dispositivo de *swap* sino que los fallos de página se servirán directamente del fichero ejecutable. El diagrama de estados correspondiente a una página de código sería el siguiente:



Donde las transiciones se corresponden con los siguientes eventos e implican las siguientes operaciones:

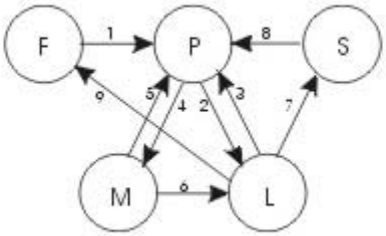
1. Evento **F**: Se trae la página del ejecutable (**F**) a memoria principal (**P**). Se produce un acceso de lectura al fichero ejecutable.

2. Evento **XT**: El algoritmo de reemplazo le quita la página al proceso pasándola al almacenamiento intermedio. Puesto que las páginas de código no se modifican, pasará a la lista de páginas libres (**L**).

3. Evento **F**: Se recupera la página de la lista de páginas libres del almacenamiento intermedio (**L**). Gracias a la técnica del almacenamiento intermedio se ha podido servir un fallo de página sin realizar operaciones de E/S al disco.

4. Evento **XP**: Se elimina la página del almacenamiento intermedio liberando el marco para que pueda usarse para otra página. Se transita al estado **F** ya que el siguiente fallo de página deberá servirse del fichero ejecutable.

b) Una página de datos con valor inicial se servirá la primera vez desde el ejecutable y posteriormente desde el dispositivo de *swap* , ya que las modificaciones que se produzcan sobre la página no deben reflejarse en el fichero ejecutable. En realidad, se podrían servir del ejecutable varios fallos asociados a una misma página de este tipo mientras que dicha página no haya sido modificada conservando el mismo contenido inicial. El diagrama de estados correspondiente a una página de este tipo sería el siguiente:



Donde las transiciones se corresponden con los siguientes eventos e implican las siguientes operaciones:

1. Evento **F**: Se trae la página del ejecutable (**F**) a memoria principal (**P**). Se produce un acceso de lectura al fichero ejecutable.

2. Evento **XT** y la página no se ha modificado (bit **M** igual a 0): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas libres (**L**).

3. Evento **F**: Se recupera la página de la lista de páginas libres del almacenamiento intermedio (**L**).

4. Evento **XT** y la página se ha modificado (bit **M** igual a 1): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas modificadas (**M**).

5. Evento **F**: Se recupera la página de la lista de páginas modificadas del almacenamiento intermedio (**M**).

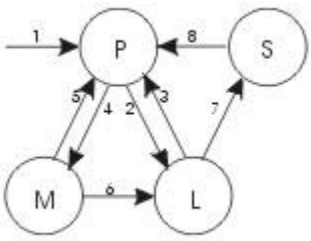
6. Evento **W**: Las páginas que están en la lista de modificadas se van escribiendo de forma agrupada al *swap*. Este evento indica que ha finalizado la operación de escritura en el disco de una determinada página y que, por lo tanto, debe pasar a la lista de páginas libres (**L**). Nótese que, al tratarse de un sistema sin preasignación de *swap*, la primera vez que se produzca esta situación, no habrá una copia en el disco para esa página por lo que en ese momento habrá que reservar espacio para la misma.

7. Evento **XP** y hay copia en *swap* de la página: Se elimina la página del almacenamiento intermedio liberando el marco para que pueda usarse para otra página. Se transita al estado **S** ya que el siguiente fallo de página deberá servirse de la copia en el dispositivo de *swap*. Nótese que en este momento no hay que escribir sobre la copia en swap ya que dicha copia se actualizó cuando esta página pasó por la lista de páginas modificadas (**M**).

8. Evento **F**: Se trae la página del dispositivo de *swap* (**S**) a memoria principal (**P**). Se produce un acceso de lectura a dicho dispositivo.

9. Evento **XP** y no hay copia en *swap* de la página: La no existencia de una copia de la página en *swap* indica que la página no se ha modificado y, por lo tanto, su contenido es idéntico al inicial por lo que el siguiente fallo de página podría servirse del ejecutable. Esta transición implica la eliminación de la página del almacenamiento intermedio y el cambio al estado **F**.

c) Una página que pertenece a la región de datos sin valor inicial no está almacenada en el ejecutable. Únicamente aparece en la cabecera del mismo el tamaño de dicha región. Cuando se produce el primer fallo de página, se busca un marco libre y, por razones de confidencialidad de la información, se rellena con ceros. En el caso de que dicha página rellena con ceros sea expulsada sin modificarse (situación poco probable pero posible), habría que reservarla espacio de *swap* puesto que necesita un lugar para almacenarse al no estar incluida en el ejecutable. El diagrama resultante sería muy similar al del apartado anterior y presentaría el siguiente aspecto:



Donde las transiciones se corresponden con los siguientes eventos e implican las siguientes operaciones:

1. Evento **F**: Se busca un marco libre en memoria principal (**P**) y se rellena con ceros.

2. Evento **XT** y la página no se ha modificado (bit **M** igual a 0): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas libres (**L**).

3. Evento **F**: Se recupera la página de la lista de páginas libres del almacenamiento intermedio (**L**).

4. Evento **XT** y la página se ha modificado (bit **M** igual a 1): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas modificadas (**M**).

5. Evento **F**: Se recupera la página de la lista de páginas modificadas del almacenamiento intermedio (**M**).

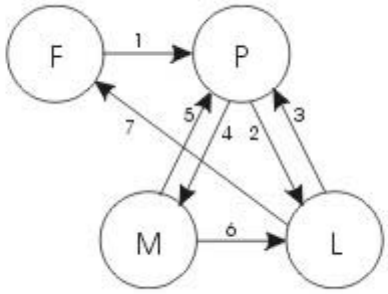
6. Evento **W**: La página pasa de la lista de modificadas (M) a la lista de páginas libres (**L**). Nótese que, al tratarse de un sistema sin preasignación de *swap*, la primera vez que se produzca esta situación, no

habrá una copia en el disco para esa página por lo que en ese momento habrá que reservar espacio para la misma.

7. Evento **XP**: Se elimina la página del almacenamiento intermedio liberando el marco para que pueda usarse para otra página. Se transita al estado **S** ya que el siguiente fallo de página deberá servirse de la copia en el dispositivo de *swap*. Nótese que en este momento, a diferencia del apartado anterior, podría ser necesario una operación de escritura en el disco. Esta situación se produciría en el caso de que la página expulsada no tenga copia en el *swap*. Se trataría de una página que se trajo a memoria con el primer fallo y se la expulsa sin haber sido modificada.

8. Evento **F**: Se trae la página del dispositivo de *swap* (**S**) a memoria principal (**P**). Se produce un acceso de lectura a dicho dispositivo.

d) Una página de un fichero proyectado no utilizará el dispositivo de swap ya que se pretende que los cambios que se hagan sobre la página queden reflejados en el fichero. El diagrama de estados correspondiente a una página de este tipo sería el siguiente:



Donde las transiciones se corresponden con los siguientes eventos e implican las siguientes operaciones:

1. Evento **F**: Se trae la página del fichero (**F**) a memoria principal (**P**). Se produce un acceso de lectura al fichero.

2. Evento **XT** y la página no se ha modificado (bit **M** igual a 0): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas libres (**L**).

3. Evento **F**: Se recupera la página de la lista de páginas libres del almacenamiento intermedio (**L**).

4. Evento **XT** y la página se ha modificado (bit **M** igual a 1): El algoritmo de reemplazo le quita la página al proceso pasándola a la lista de páginas modificadas (**M**).

5. Evento **F**: Se recupera la página de la lista de páginas modificadas del almacenamiento intermedio (**M**).

6. Evento **W**: Este evento indica que ha finalizado la operación de escritura en el fichero de la página y que, por lo tanto, debe pasar a la lista de páginas libres (**L**).

7. Evento **XP**: Se elimina la página del almacenamiento intermedio liberando el marco para que pueda usarse para otra página. Se transita al estado **F** ya que el siguiente fallo de página deberá servirse del fichero. la copia en el dispositivo de *swap*. Nótese que en este momento no hay que escribir sobre el fichero ya que esta operación se realizó cuando esta página pasó por la lista de páginas modificadas (**M**)

Ejercicio 3

Sea un sistema con memoria virtual y paginación por demanda, que emplea direcciones de 32 bits y páginas de 1 KB. Sobre este sistema se desea ejecutar el siguiente programa:

```
char vector [15000];
int a = 5; int b = 15; int c = 23;
int x; char d;

main (void)          {
    int pid, j, fd;
```

```
        char *w, *y, *k;

        x = a;
A
        w = (char *)malloc (18000);
B
        pid = fork();
C
        if (pid == 0) {
            for (j = 0; j < 15000; j ++)
                vector[j] = 'a';
        }
        else {
            fd = open("datos.txt", O_WRONLY);
            y = (char *)mmap(0,15000,PROT_WRITE,MAP_SHARED,fd,0);
            k = y;
D
            for (j = 0; j < 15000; j ++){
                *k = vector[j];
                k++;
            }
            munmap(y);
        }
E
        free(w);
F
        .
        .
        exit (0);
    }
```

El código del mencionado programa ocupa 21 KB y se utilizan bibliotecas estáticas, ocupando el código de todas las funciones de biblioteca 1 página. Además el tamaño de la pila nunca excede de 1 página. Conteste razonadamente a las siguientes cuestiones:

a) Indique las distintas regiones que configuran la imagen de memoria del proceso así como su tamaño en los siguientes puntos: justo antes de ejecutar la función main, A, B, C, D, E y F.

b) Para cada una de las regiones del proceso indique, para los mismos puntos anteriores, el soporte utilizado por la región en memoria secundaria, el número de páginas que residen en dicho soporte, el número de páginas en memoria principal y el número de páginas que habría que escribir a memoria secundaria en caso de que el proceso fuera expulsado de memoria principal. Responda a estas cuestiones ***para cada uno*** *de los siguientes gestores de memoria:*

- *Gestor de memoria que preasigna espacio de swap sólo para las regiones sin soporte propio y no utiliza copy-on-write*
- *Gestor de memoria que no preasigna espacio de swap y* ***no*** *emplea copy-on-write*
- *Gestor de memoria que no preasigna espacio de swap y emplea copy-on-write*

Nota*: Considere que hay suficientes marcos de página en memoria principal.*

Solución

a) La imagen de un proceso está formada por un conjunto de direcciones virtuales, generalmente no contiguas. Esta imagen de memoria se divide en un conjunto de *regiones* lógicas o *segmentos*, que se pueden definir como una zona contigua del espacio de direcciones virtual del proceso que se puede tratar como una unidad u objeto distinguible a la hora de ser compartida o protegida. A continuación se indican las distintas regiones que configuran la imagen de memoria en los distintos puntos.

Antes de main:

- *Código*, ocupa 22 páginas (el código del programa ocupa 21 KB, es decir, 21 páginas y el código de las funciones de biblioteca 1 página).
- *Datos con valor inicial (DVI),* esta región almacena el contenido de las variables globales con valor inicial (las variables a, b y c). Esta región cabe perfectamente en una página.
- *Datos sin valor inicial (DSVI)*, esta región almacena el contenido de las variables globales que no tienen valor inicial (vector, x y f). Todas estas variables ocupan 15000 + 4 + 1 = 15005 bytes, por lo que se necesitan 15 páginas.
- *Pila*, esta región antes de main almacena el entorno del proceso y los parámetros que se pasan a la función main. Según el enunciado, esta región no excede de 1 página y por lo tanto en este punto ocupará 1 página.

Recuerde que una región está constituida por una serie de páginas virtuales, por tanto las páginas anteriores hacen referencia a páginas virtuales. Estas páginas pueden residir en memoria principal, en swap o en ningún sitio (por ejemplo, las páginas correspondientes a datos sin valor inicial no residen inicialmente en ningún sitio en caso de utilizar paginación por demanda y un gestor que no preasigne espacio de swap. En este caso las páginas se crean en memoria principal, rellenandose a ceros, la primera vez que se accedan).

Punto A:

La imagen de memoria es la misma que la del apartado anterior. En este punto se ha modificado la pila, ya que se ha incluido en ella el registro de activación para la función main, que incluye la dirección de retorno, los parámetros pasados (en este caso ninguno) y las variables locales definidas en main. Su tamaño sigue siendo de 1 página. También se ha modificado una página de DSVI, aquella donde se almacena el valor de la variable x.

Punto B:

La función malloc reserva 18000 bytes. La dirección de comienzo de esta zona se almacena en la variable w, que se encuentra en la pila. Este espacio se reserva en la zona de datos sin valor inicial, por lo tanto, su tamaño se incrementa en este punto a 15005 + 18000 = 33005 bytes, por lo que se necesitan 33 páginas. Observe que éstas son páginas virtuales que no se han cargado en memoria, ya que se utiliza paginación por demanda.

Para añadir este espacio al proceso basta con modificar su tabla de páginas añadiendo tantas entradas como hagan falta. Estas entradas se marcarán como no presentes. Si se utiliza preasignación de espacio de swap será necesario reservar espacio en swap. En caso contrario no hará falta reservar espacio en ninguna parte (ni en memoria principal ni en swap). Cuando se acceda a ellas por primera vez, se producirá un fallo de página y se reservará un marco en memoria principal, que se rellenará a ceros.

Punto C:

En este punto se crea un proceso hijo, cuya imagen de memoria es una copia de la del padre.

Punto D:

El proceso hijo mantiene la misma imagen de memoria. El padre, sin embargo, proyecta un fichero en memoria. Por lo tanto se crea una nueva región correspondiente a un *fichero proyectado en memoria (FPM)*. Esta región ocupa 15000 bytes, es decir 15 páginas. De nuevo, basta con modificar la tabla de páginas del proceso, añadiendo tantas entradas como sea necesario (15 en este caso). Estas entradas se marcarán como no presentes y se traerán del fichero a medida que se vayan referenciando.

Punto E:

El hijo mantiene la misma imagen de memoria. El padre desproyecta el fichero de memoria y por lo tanto esta región desaparece de su imagen de memoria. Para ello se invalidan las entradas de la tabla de páginas correspondiente a estas región. También es necesario invalidar cualquier entrada que se encuentre en la TLB.

Punto F:

Padre e hijo liberan la memoria reservada en el punto B. Por lo tanto se reduce la región de datos sin valor inicial, que de nuevo vuelve a tener 15 páginas.

b) La región de código utiliza como soporte el propio fichero ejecutable. Las páginas se irán trayendo de él a medida que se referencien. Los datos con valor inicial tienen como soporte inicial el propio fichero

ejecutable. Siempre que no se modifiquen las páginas correspondientes a esta región, los fallos de página podrán servirse del fichero ejecutable (en este caso nunca se modifican las páginas de DVI). En caso de modificar una página de este tipo se utilizará como soporte el área de swap. Los datos sin valor inicial y la pila no tienen un soporte propio en el sistema de ficheros y utilizan el swap. Por último, las regiones correspondientes a ficheros proyectados en memoria utilizan como soporte el propio fichero que se proyecta.

En las secciones siguientes se utilizará PS para indicar las páginas en soporte, PM para las páginas en memoria principal y sucias para indicar las páginas que se han modificado y que deberían escribirse en el soporte utilizado en memoria secundaria en caso de ser expulsadas de memoria principal.

Gestor de memoria que preasigna espacio de swap para las regiones sin soporte y no emplea copy-on-write

En este caso se reserva espacio de swap para las regiones sin soporte propio (DSVI, pila y DVI modificados).

Antes de main

Región	PS	PM	sucias
Código	22	0	0
DVI	1	0	0
DSVI	15	0	0
Pila	1	1	1

La pila se crea inicialmente en memoria principal para almacenar el entorno de proceso y los parámetros pasados al proceso.

Punto A:

Región	PS	PM	sucias
Código	22	1	0
DVI	1	1	0
DSVI	15	1	1
Pila	1	1	1

Se empieza a ejecutar el programa y se produce el fallo de la página de código correspondiente. Según el enunciado, es fácil ver que el código que se presenta de la función **main** cabe en un página. Se accede y se modifica la página donde se almacena `x` (una página de la región de DSVI). También se accede a una página de DVI, aquella donde se almacena `a`.

Punto B:

Región	PS	PM	sucias
Código	22	2	0
DVI	1	1	0
DSVI	33	1	1
Pila	1	1	1

Se ejecuta el código de la función malloc, que se almacena en una página. Su ejecución provoca una fallo de página, que se sirve del fichero ejecutable, ya que se utilizan bibliotecas estáticas. Se añade espacio en swap para la memoria dinámica que se ha reservado.

Punto C:

Podemos suponer que se comparten en memoria principal las páginas de código. Como no se emplea *copy-on-write* es necesario copiar el resto de páginas del padre al hijo reservando espacio también en swap. Padre e hijo, por lo tanto, tendrán la configuración anterior

Punto D:

Proceso Padre				**Proceso Hijo**			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	33	1	1	DSVI	33	15	15

Pila	1	1	1	Pila	1	1	1
FPM	15	0	0				

El hijo modifica la variable `vector`. Se irán reservando en memoria principal marcos de página que se rellenarán inicialmente a ceros. El padre proyecta un fichero en memoria. Observe que no se carga ninguna página del fichero en memoria principal. Esto se debe a que se utiliza paginación por demanda.

Punto E:

Proceso Padre				**Proceso Hijo**			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	33	1	1	DSVI	33	15	15
Pila	1	1	1	Pila	1	1	1

Se desproyecta el fichero en el proceso padre y se liberan las páginas que se cargaron en memoria principal en el bucle `for` anterior a este punto.

Punto F:

Proceso Padre				**Proceso Hijo**			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	15	1	1	DSVI	15	15	15
Pila	1	1	1	Pila	1	1	1

Gestor de memoria que no preasigna espacio de swap y no emplea copy-on-write

En este caso no se preasigna espacio de swap. Este espacio se reservará cuando haya que expulsar una página de memoria principal.

Antes de main

Región	**PS**	**PM**	**sucias**
Código	22	0	0
DVI	1	0	0
DSVI	0	0	0
Pila	0	1	1

En este caso no se reserva espacio en swap para la pila ni para los DSVI. Los fallos de página para estas regiones se servirán inicialmente reservando marcos de página que se rellenarán a ceros.

Punto A:

Región	**PS**	**PM**	**sucias**
Código	22	1	0
DVI	1	1	0
DSVI	0	1	1
Pila	0	1	1

Punto B:

Región	**PS**	**PM**	**sucias**
Código	22	2	0
DVI	1	1	0
DSVI	0	1	1
Pila	0	1	1

Punto C:

Podemos suponer que se comparten en memoria principal las páginas de código. Como no se emplea *copy-on-writ*e es necesario copiar el resto de páginas del padre al hijo reservando espacio también en swap. Padre e hijo, por lo tanto, tendrán la configuración anterior

Punto D:

Proceso Padre				Proceso Hijo			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	0	1	1	DSVI	0	15	15
Pila	0	1	1	Pila	0	1	1
FPM	15	0	0				

Punto E:

Proceso Padre				Proceso Hijo			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	0	1	1	DSVI	0	15	15
Pila	0	1	1	Pila	0	1	1

Punto F:

Proceso Padre				Proceso Hijo			
Región	**PS**	**PM**	**sucias**	**Región**	**PS**	**PM**	**sucias**
Código	22	2	0	Código	22	2	0
DVI	1	1	0	DVI	1	1	0
DSVI	0	1	1	DSVI	0	15	15
Pila	0	1	1	Pila	0	1	1

Gestor de memoria que no preasigna espacio de swap y emplea copy-on-write

Este caso es similar al anterior. La única diferencia es que cuando se ejecuta la llamada fork se comparten inicialmente todas las páginas, marcando como copy-on-write aquellas correspondientes a datos y pila. Cuando un proceso acceda en escritura a una página de éstas, se producirá un trap y el sistema operativo hará una copia de la página.