CONCEPTOS IMPORTANTES

Un computador tiene cierta memoria principal que utiliza para mantener los programas en ejecución. En los computadores modernos se pueden colocar varios programas en memoria al mismo tiempo.

En el caso más simple, la máxima cantidad de espacio de direcciones que tiene un proceso es menor que la memoria principal.

Hoy en día existe una técnica llamada **Memoria Virtual** en la que el SO mantiene una parte del espacio de direcciones en memoria principal y otra parte en disco

Memoria virtual

La memoria virtual proporciona la habilidad de ejecutar programas más extensos que en la memoria física de la computadora (que la RAM), trayendo pedazos entre la RAM y el disco. También permite ligar dinámicamente bibliotecas en tiempo de ejecución.

Llamadas al sistema

FUNCIONES DEL SISTEMA OPERATIVO

- Proveer abstracciones a los programas de usuario
- Administrar recursos de la computadora

La **llamada al sitema** generalmente es el mecanismo usado por una aplicación para solicitar un servicio al sistema operativo. Normalmente usan una instrucción especial de la CPU.

Cuando una llamada al sistema es invocada, la ejecución del programa es interrumpida y sus datos son guardados para poder ejecutarse luego.

Para hacer más entendible el mecanismo de llamadas al sistema, vamos a echar un vistazo a la llamada ***read***

```
cuenta = read(fd, buffer, nbytes);
```

La llamada devuelve le nº de bbytes que se leen en *cuenta* (por lo general el valor de *cuenta* es el mismo que *nbytes* pero puede ser menor si se encuentra el fin del archivo al estar leyendo).

Si hay un error *cuenta* se establece a '-1' y se coloca el error en '*errno*'

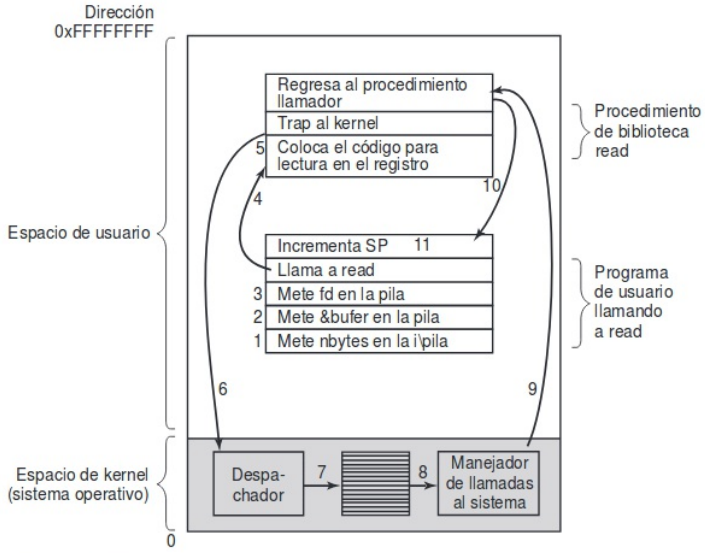


Figura 1-17. Los 11 pasos para realizar la llamada al sistema `read(fd, bufer, nbytes)`.

Centrandonos más en el ejemplo...

1. Se pasan los parámetros de la función como si fuera una pila (de ahí que estén en orden inversa). Además el segundo parámetro se pasa como referencia, al contrario que los otros, que se pasan como valor. Por esa razón hay que indicar 'contenido de bufer' con el &.
2. Lo hace con buffer.
3. Lo hace con fd.
4. Llamada al procedimiento de biblioteca (típica llamada a procedimiento)
5. La biblioteca coloca por lo general el nº de la llamada al sistema en un lugar en el que el SO lo espera, como por ejemplo, y, en este caso, un registro.
6. Se ejecuta un TRAP para cambiar a modo kernel. Aunque la instrucción TRAP no salta a una dirección donde se encuentre el procedimiento.
7. Por lo que salta a una dirección arbitraria donde hay un campo de 8 bits que proporciona un índice a una tabla en memoria que le indica el salto que tiene que hacer para continuar con el procedimiento.
8. Se le proporciona el índice al manejador de llamadas para devolver el procedimiento a la biblioteca.
9. Ejecuta el manejador de llamadas que le devuelve al procedimiento de biblioteca (otra vez espacio usuario).
10. Por último regresa en la forma usual a la que regresan las llamadas a procedimientos.

Por último el programa usuario limpia la pila.

Preludio

El trabajo del SO es abstraer una jerarquía de memoria en un modelo útil y administrarla; la parte que hace esto se llama **Administrador de memoria**. En este capítulo nos concentraremos en el modeo del programador de la memoria principal y cómo se puede administrar.

Sin Abstracción de Memoria

Cada programa veía simplemente la memoria física. Cuando un programa veía la instrucción

```
MOV REGISTRO1, 1000
```

la computadora movía el contenido ***100*** a ***REGISTRO1*** dentro de la memoria física.

De esta forma no había manera de tener 2 programas ejecutándose a la vez. Ya que se borrarían mutuamente si intentaran acceder a la misma dirección mientras están en ejecución.

Los modelos (a) y (c) tienen la desventaja de que un error en el programa de usuario puede borrar el SO (al estar en memoria de escritura):

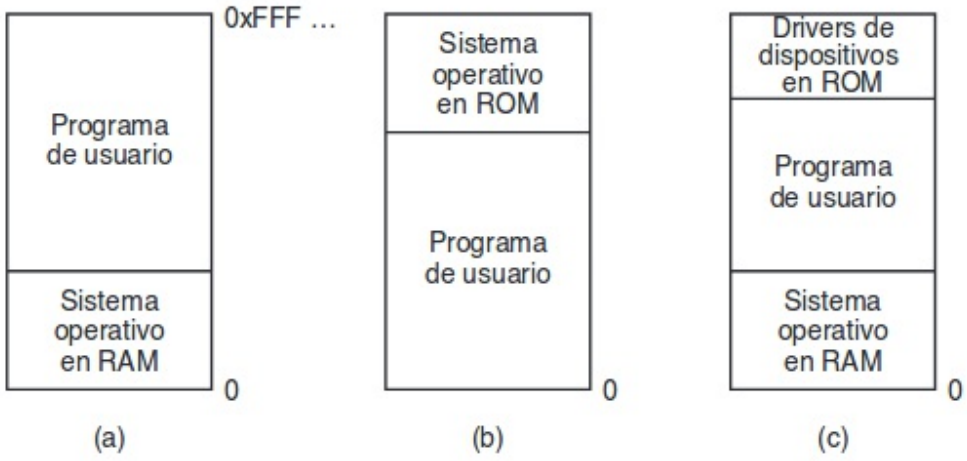


Figura 3-1. Tres formas simples de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

Ejecución de múltiple programas sin abstracción

El programa guarda todo el contenido de la memoria en un archivo en disco, más tarde cuando vuelve al programa lo vuelve a traer.

Sin embargo esto trae un problema como se puede comprobar en la imagen...

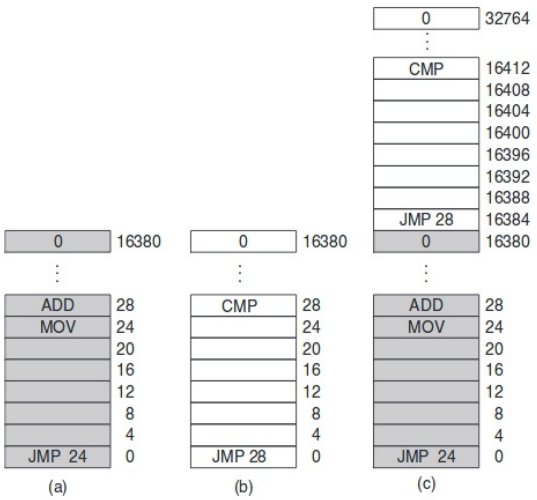


Figura 3-2. Ilustración del problema de reubicación. (a) Un programa de 16 KB. (b) Otro programa de 16 KB. (c) Los dos programas cargados consecutivamente en la memoria.

El primero de los programas tiene una llave de memoria diferente al segundo para poder distinguirlos. Al

principio el programa salta a la dirección 24; el segundo salta a la 28. Cuando los dos programas se cargan consecutivamente en memoria empezando en dirección 0.

Con esto, los programas no se interfieren porque tienen distinta llave, pero las instrucciones (por ejemplo en los JMP) saltan a direcciones que no deberían saltar. Eso es porque los programas se están refiriendo a memoria física.

Una solución sería utilizar la **reubicación estática** que consiste en sumarle el valor corresponiente a la posicion en la en disco que se le asocia. Es decir, si un programa se cargaba en la dirección 16,384, se sumaba ese mismo valor.

Una Abstracción de memoria: Espacios de direcciones

Una solución para permitir que haya varias aplicaciones en memoria al mismo tiempo es la usada anteriormente, con la llave que indica qué instrucción pertenece a cada programa.

La otra es crear la abstracción del espacio de direcciones.

Un **espacio de direcciones** es el conjunto de direcciones que puede utilizar un proceso para direccionar la memoria. Cada proceso tiene su propio espacio de direcciones. Lo complicado es proporcionar a cada programa su espacio de manera que la dirección 28 de un programa indique la ubicación física distinta de la 28 en otro programa.

Registros base y límite

La solución sencilla utiliza una versión simple de **reubicación dinámica**. Asocia el espacio de direcciones de cada proceso sobre una parte distinta de la memoria física. Esta solución contiene dos registros de hardware especiales **base** y **límite**. Cuando se utilizan estos registros, van aumentando, es decir, van acotando los límites que pueden tener los demás programas para evitar espacios usados.

Intercambio

Si la memoria física de la computadora es lo bastante grande para contener todos los procesos, los esquemas descritos hasta ahora funcionarán correcramente. Pero en la práctica la RAM es un recurso muy preciado por todos los programas.

La estrategia que se ha desarrollado con los años consiste en utilizar el **intercambio** que consiste en **llevar cada proceso completo a memoria, ejecutarlo durante un tiempo y regresarlo a disco**.

La otra estrategia es conocida como **MEMORIA VIRTUAL**

La operación de intercambio se describe visualmente así:

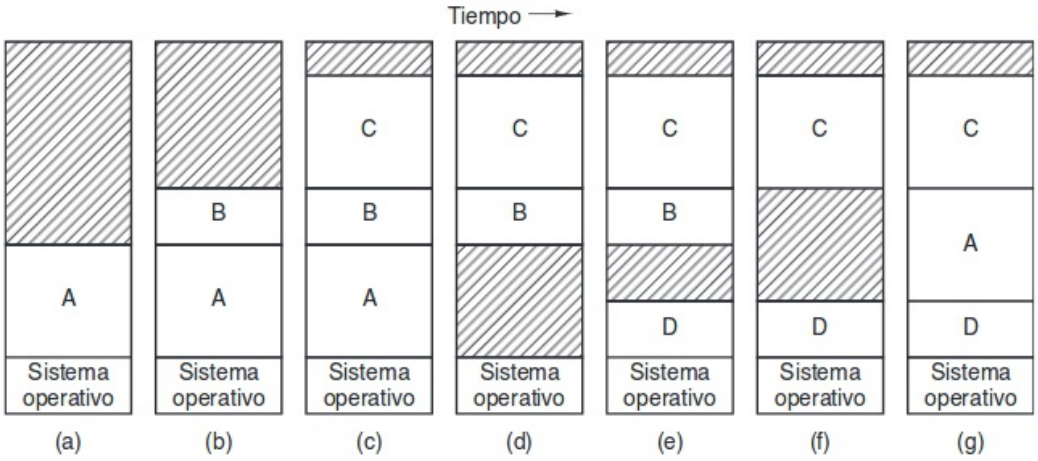


Figura 3-4. La asignación de la memoria cambia a medida que llegan procesos a la memoria y salen de ésta. Las regiones sombreadas son la memoria sin usar.

Administración de memoria libre

Administración con mapas de bits

La memoria se divide en unidades de asignación (pueden ser del tamaño que convenga), en cada unidad hay un bit correspondiente en el mapa de bits (0 libre y 1 ocupada)

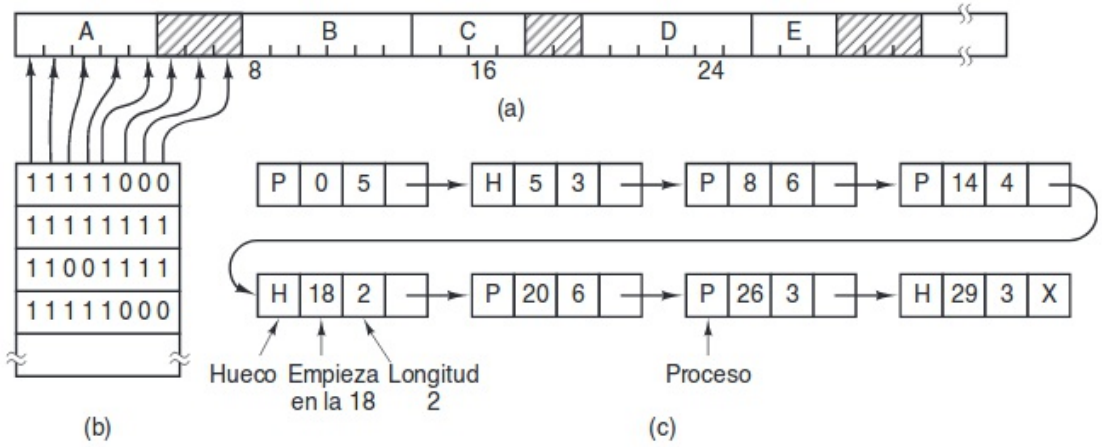


Figura 3-6. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas de graduación muestran las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

Administración con listas ligadas

Muy sencillo el concepto... se explica en la imagen anterior (c).

Cuando los procesos y huecos se mantienen en una lista ordenada por dirección, se pueden utilizar varios algoritmos para asignar memoria.

1. **Primer ajuste**: se explora la lista hasta encontrar un hueco lo suficientemente grande y se divide en 2: uno para el proceso y otro para su memoria sin utilizar.
2. **Siguiente ajuste**: igual que el **primer ajuste** pero se lleva un registro de los huecos. (rendimiento más

pobre... mucha pérdida computacional).

3. **Mejor ajuste**: busca en toda la lista y toma el hueco más pequeño que sea adecuado. En vez de hacer como el primero y buscar también hueco para la memoria sin utilizar aún de es proceso, direcamente busca el tamaño actual necesario. (es peor que los 2 anteriores).
4. **Peor ajuste**: toma siempre el hueco más grande disponible.
5. **Ajuste rápido**: en vez de mantener una misma lista para los procesos y huecos, súnicamente se mantiene una para los huecos y sorprendentemente es más óptimo.

1. Introducción

El método llamado Memoria Virtual se basa en la idea de dividir programas en sobrepuestos (*Overlays*) de tal forma que se mantienen en disco y se intercambian hacia dentro de la memoria.

Aplicando el concepto a Memoria Virtual

Cada programa tiene su espacio de direcciones, que se divide en trozos llamados **páginas**.

Cada página es un rango continuo de direcciones. Las páginas se asocian a la memoria física (aunque no es necesario estar en la memoria física para ejecutar el programa).

Con la **memoria virtual** todo el espacio de direcciones (incluido datos y texto) se puede asociar a la memoria física.

```
Memoria Física = RAM
```

2. Paginación

Cuando un programa ejecuta una instrucción como ***MOV REG, 100*** copia el contenido de la dirección de memoria ***1000*** a ***REG*** generando así direcciones virtuales.

Estas direcciones virtuales generadas por el programa se conocen como **direcciones virtuales** y forman el **espacio de direcciones virtuales**. (Si no hubiera memoria virtual, se colocaría directamente la memoria física en el bus de memoria y se lee/escribe directamente en esa dirección).

Cuando se utiliza una memoria virtual, las direcciones virtuales no van directamente al bus de memoria: van al **MMU** (*Memory Management Unit*) que asocia direcciones virtuales a direcciones físicas.

```
Direcciones Virtuales de 16 bits --> hasta 64KB
Memoria Física --> 32KB

El espacio de direcciones VIRTUALES se divide en PÁGINAS --> 4KB
El espacio de direcciones FÍSICAS se divide en MARCOS DE PÁGINA --> 4KB
    *PAGINA y MARCO DE PÁGINA tienen mismo TAMAÑO
```

```
Por lo tanto obtenemos 16 PÁGINAS y 8 MARCOS DE PÁGINA
```

Imagen: Paginacion 1

Cuando el programa ejecuta ***MOV REG,0*** ... trata de acceder a la dirección virtual 0. Esta dirección virtual se envía a la MMU. La MMU ve que está en la página 0 (de 0-4095), que esta ya asociada al marco de página 2 (de 8192-12287). Por lo que el bus recibe la dirección **8192**.

Como se puede comprobar en la imagen, sólo 8 de las páginas virtuales se asocian a la memoria física. En hardware, existe un **bit de presente/ausente** que lleva el registro de las páginas presentes en Memoria Virtual.

Fallo de página

Cuando el programa ejecuta ***MOV REG,32780*** está accediento a un byte dentro de la página virtual 8.

La MMU detecta que la página no está asociada y hace que la CPU haga un **trap** (***fallo de página***). El SO selecciona un marco de página que se utilize poco y escribe su contenido de vuelta al disco, obtiene la página y reinicia la instrucción del trap.

Ahora veamos un ejemplo de cómo funciona la MMU...

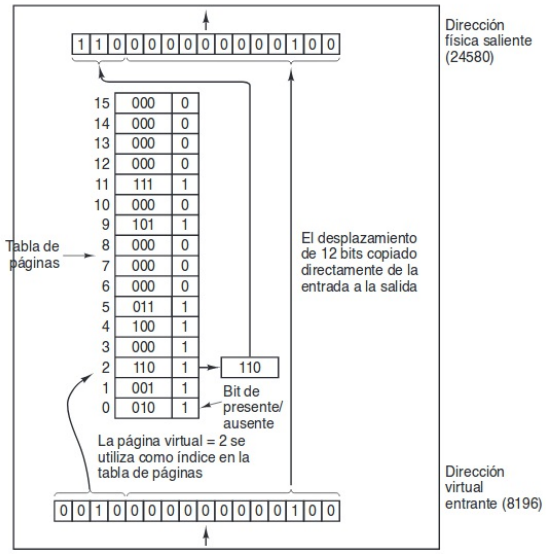


Figura 3-10. La operación interna de la MMU con 16 páginas de 4 KB.

```
Tenemos una dirección virtual: 8196 (0010000000000100)
La MMU va a asociar esta dirección con la memoria física.

1.  16 bits de entrada > 4 bits para el nº de página (16 páginas)
                       > 12 bits para desplaz. (4096 bytes dentro de una pág)
2.  Los 4 bits identifican las páginas (ordenadas) y cada página tiene dentro
un índice al marco de página físico (con su bit presente/ausente)
3.  Se obtiene el marco de página y se le suma el desplaz.

En total obtenmos de una dirección virtual de 16 bits a una física de 15.
```

```
Finalmente el conjunto de 15 bits se coloca en el bus de salida como dirección
de memoria física.
```

Tablas de páginas

El objetivo de las **tablas de página** es asociar las direcciones virtuales a las direcciones físicas. En sentido matemático, la tabla es una función donde el número de página virtual es un argumento y el número de marco físico es un resultado.

Estructura de tablas

La distribución exacta depende en gran parte de la máquina, pero en rasgos generales siempre es igual.

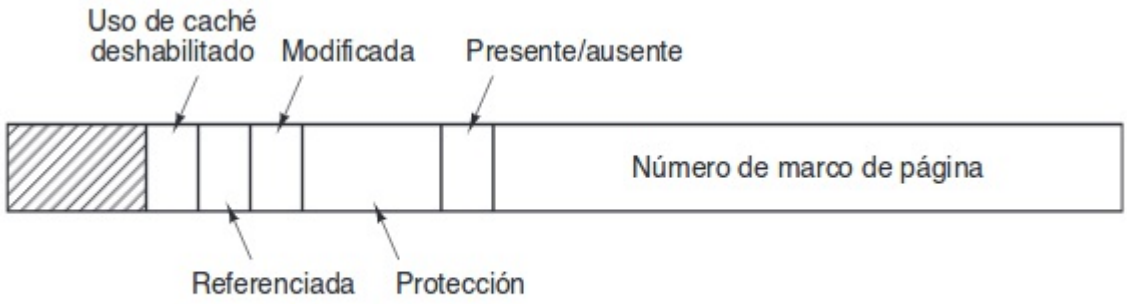


Figura 3-11. Una típica entrada en la tabla de páginas.

```
El tamaño puede variar pero 32 bits es muy común.

1.  Número de marco de Página
2.  Bit presente/ausente
3.  Protección
4.  Modificada (a veces es le refiere como 'bit sucio')
5.  Referenciada
6.  Uso de caché deshabilitado

*Los bits de Modificada son útiles a la hora de llevar un registro de páginas;
si ha sido modificada debe escribirse de vuelta en el disco.

*Los bits de Referenciada son útiles a la hora de saber qué página se debe
desalojar cuando ocurre un fallo de página. Las páginas que no se estén
utilizando son buenas candidatas.
```

Optimización de paginación

La mayor parte de las técnicas de optimización es que la tabla de páginas esté en la memoria física. Esta última idea se basa en que la mayor parte de los programas tienden a hacer un gran número de referencias a un pequeño número de páginas (se utiliza una pequeña fracción de la entrada de las páginas).

La solución que se ha ideado es equipar a las computadoras con un pequeño dispositivo de hardware, el cual

se usa para acceder a direcciones físicas sin pasar por la tabla de páginas (es decir, sin pasar 2 veces por RAM).

TLB

El dispositivo del que acabamos de hablar se llama **TLB** o **Memoria Asociativa**, éste se encuentra en la MMU y tiene el mismo nº de datos que proporciona la **tabla de páginas** + *nº página virtual*

Funcionamiento

Cuando se presenta una dirección virtual a la MMU, el hardware comprueba si el número de página está presente en TLB comparándola con todas sus entradas simultáneamente (por eso está en hardware)

1. Si está presente y **no** viola los bits de protección --> MARCO
2. Si está presenta y **sí** viola los bits de protección --> FALLO (por protección)
3. Si **no** está presente. La MMU lo detecta y la busca en la tabla de páginas, remplazándola en una de las entradas de la TLB.

El beneficio de la TLB es que ahorra área de CACHÉS DE CPU

OPTIMIZACIÓN: El Sistema Operativo intenta adivinar qué páginas tienen más probabilidad de ser utilizadas y las precarga en el TLB

Fallos de TLB

- **Fallo suave**: la página no está en TLB pero sí en Memoria Virtual.
- **Fallo duro**: la página ni siquiera está en Memoria Virtual.

Tablas de páginas Multinivel

Para estudiar este caso es necesario tener a mano esta imagen.

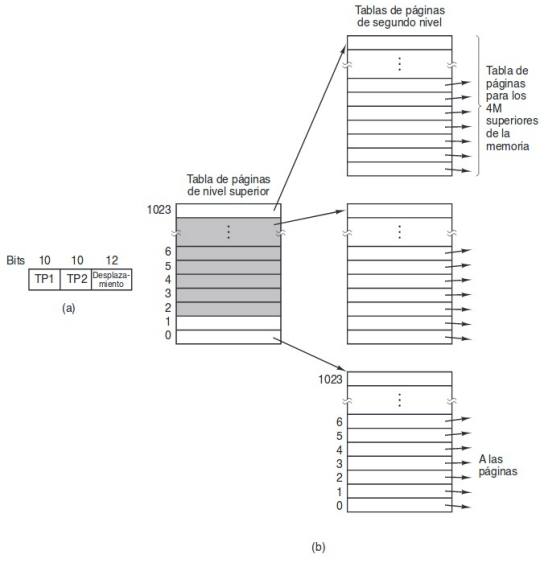


Figura 3-13. (a) Una dirección de 32 bits con dos campos de tablas de páginas. (b) Tablas de páginas de dos niveles.

```
Tenemos una dirección virtual de 32 bits:
> TP1 = 10  +  TP2 = 10  +  Despl. = 12
```

```
Desplazamiento = 12 bits --> 2¹² = 4096 Bytes = páginas de 4KB
                                  y hay un total de 2²⁰
    (ya que el despl. indica los bytes escogidos dentro de un marco)


TP1 = 10 bits --> 2¹⁰ = 1024 Bytes = 1042 entradas
                  y cada entrada es una página y cada una contiene sus 4KB

A la Tabla de Páginas de Nivel Superior se accede según el TP1


TP2 = 10 bits --> 2¹⁰ = 1024 Bytes = 1042 entradas
                  y cada entrada es una página y cada una contiene sus 4KB

Ahora después de elgir una Tabla de Nivel Superior, vamos a elegir la entrada
de esa tabla para elegir una página que nos conduzca a las últimas tablas.

Para eso sirve el TP2, para una vez tienes la Tabla Superior, llegar a las
últimas.


*Estudiar MUY DETENIDAMENTE fijándose en la figura 7
*Para ver un ejmplo más concreto leer página 200 de Tanenbaum
```

Tablas de páginas Invertidas

En este diseño existe una entrada por cada **marco de página** en la memoria física, en vez de tener una entrada por cada **página**. En la figura 6, se puede ver que **no** es una tabla invertida, ya que existe una entrada por cada marco y así ocupa bastante más que si fuera por cada página.

En este caso la entrada lleva un registro de quién se encuentra en el marco de página.

```
Direcciones Virtuales   64 bits                 -->  en el bus de entrada hay
64 bits
Página                  4 KB = 2¹² = 4096 Bytes  -->  2⁵² bits para elegir el
índice de página
RAM                     1 GB

POR COMPLETAAAAAAAAAR ES MUUUUUUUUUUUUUUUUUUUUUUUUUY DIFICIL
```

Algoritmos de remplazo de Página

No es óptimo escoger una página al azar para remplazar, por eso se trata de elegir una página de uso poco frecuente.

El algoritmo utópico se basa en etiquetar cada página con el número de instrucciones que se ejecutarán

después de que se haga referencia a esa pagina. El problema es que no hay forma de saber cuál será la próxima página referenciada.

1. No usadas recientemente

La mayor parte de las computadoras tienen dos bits de estados:

- **Referencia** que se establece cada vez que se hace referencia a la página.
- **Modifica** que se establece cuando se modifica la página.

Al principio, cuando se carga por primera vez la página, sólo se carga en SÓLO LECTURA. Sin embargo, cuando se modifica y por ende se cambia el bit **Modifica** también se pone en modo LECTURA/ESCRITURA.

En el transcurso del programa el bit **R** se borra en cada interrupción de reloj. Cuando ocurre un fallo de página, el Sistema Operativo inspecciona todas las páginas y las divide en 4 categorías:

0. no R, no M.
1. no R, sí M. (ya que se ha borrado el R en una interrupción)
2. sí R, no M. (solo se ha llevado para leer)
3. sí R, sí M.

El algoritmo NRU elimina una página al azar de la mínima numeración que no esté vacía

2. Remplazo FIFO

El Sistema Operativo mantiene una lista de todas las páginas actualizadas en memoria, las páginas que llegan se ponen en la parte reciente y se van empujando hacia el otro lado.

Cuando ocurre un fallo de página, se elimina la página que esté al final (la que se ha usado hace más tiempo).

Sin embargo, aunque puede parecer perfecta, puede dar la casualidad de que una página que se usa extremadamente a menudo no se usó durante X instrucciones y la estamos eliminando por usar este algoritmo.

3. Segunda Oportunidad

Es una modificación del FIFO que evita precisamente el problema comentado en el último párrafo.

Lo que hace es analizar siempre el bit **Referenciado** del último elemento, el que se va a eliminar:

- Si el bit es 0 --> se elimina inmediatamente (hace tiempo que se ha borrado).
- Si el bit es 1 --> se pone el bit a 0 y la página se pone al final de la lista. *De ahí viene lo de **segunda oportunidad**.

4. Remplazo de Reloj

Más óptimo que mantener una lista, es mantener una con la forma de reloj para no tener que mover el puntero de detrás a delante.

Por lo demás es exactamente igual a **segunda oportunidad** es decir, si el bit **R** es 1, se pone a 0 y la manecilla

pasa al siguiente.

5. Menos usadas Recientemente (LRU)

Este algoritmo es excelente, ya que se fija en la proporción de veces que se utiliza una página. El problema es que es muy caro de implementar y mantener.

Una forma es implementarlo mediante hardware:

```
Se implementa un contador llamado C (que indica la cantidad de veces que se
ejecuta una instrucción en concreto), que se incrementa después de cada
instrucción.

En cada entrada se almacena el valor de C.

Cuando ocurre un fallo de página el SO examina todos los contadores y el menor
lo elimina.
```

Otra forma con matrices...

```
Tenemos una matriz n*n donde n es el nº de marcos de página.

Inicialmente la matriz esta en 0.

Cuando se referencia una página, se establecen todos los bits de su fila en 1
y de su columna en 0.

En cualquier momento, la fila cuyo valor binario sea menor se elimina.
```

6. Remplazo de conjunto de trabajo

Es la forma más pura de paginación: los procesos inician sin ninguna de sus páginas en la memoria. A medida que se vayan solicitando páginas, van surgiendo fallos de página.

Lo bueno es que se evita cargar páginas no necesarias al principio, de tal modo que al cabo de un cierto tiempo, después de muchos fallos de página, seguramente va a haber muy pocos fallos ya que cada programa tiende a referirse a una fracción pequeña de todas sus páginas. (Este conjunto de páginas se llama **conjunto de trabajo**).

*Nota: se dice que un programa que produce fallos de página cada pocas instrucciones está **sobrepaginando**.

Aquí se ve una gráfica de este método para entenderlo: Conjunto de Trabajo

7. Remplazo WSClock

Este algoritmo fusiona el **algoritmo del reloj** con el **conjunto de trabajo**.

Al principio existe una lista circular de marcos de página vacías. Cuando se carga una página de agrega a la lista. En cada entrada se incluye el ***tiempo de último uso***. Por lo que, al igual que en el algoritmo de reloj, se analizan los bits de **R** y **M**. Pero además se mira su edad, es decir, el tiempo ultimo uso... Si no coincide con el *tiempo actual del grupo de trabajo* y **R** es 0 (se ha eliminado R por una interrupción) se elimina.

Cuestiones de Diseño para Paginación

Asignación Local vs Global

- **Algoritmo Local**: asignan una fracción fija de memoria a cada proceso.
- **Algoritmo Global**: asignan dinámicamente la memoria a los procesos.

Lo más **óptimo** es usal el **algoritmo global** ya que cambia dinámicamente y no se desperdicia memoria.

Algunos algoritmos de remplazo pueden necesitar uno u otro, por ejemplo WSClock sólo tiene sentido con estrategia local. (dado que juega con el grupo de trabajo -> local)

Algoritmo PFF (Page Fault Frequency)

Indica cuando se debe aumentar o disminuir la asignación de marcos de página a un proceso.

Tamaño de Página

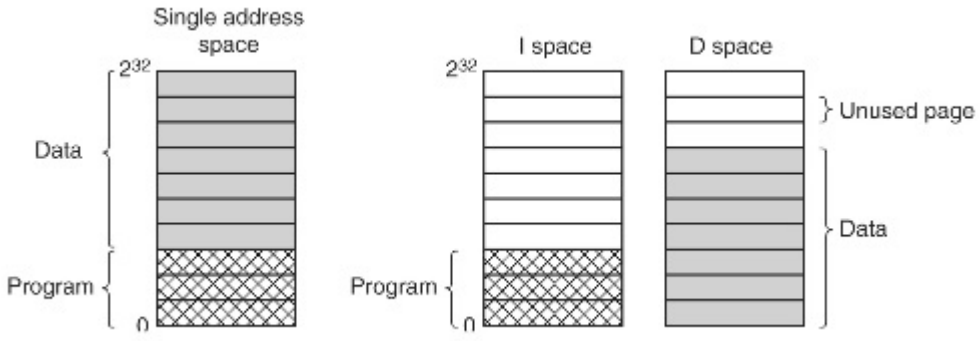
Puede ser elegido por el SO y no hay un tamaño óptimo por lo que se rigen por distintos factores:

- **Página pequeña**: Menor fragmentación interna.
- **Página grande**: La tabla de páginas es más pequeña. Pero el tiempo de transferencia de una página grande es **poco** mejor que el de una pequeña.

Espacios de instrucciones y datos

En espacios pequeños no caben instrucciones y datos; se usan espacios separados.

Lo que provoca el duplicado de espacio de direcciones como se puede ver en la imagen.



Páginas compartidas

Si varios usuarios ejecutan el mismo programa es recomandable compartir páginas en vez de varias copias en

memoria.

Las páginas de sólo **lectura** (código) son muy sencillas, sólo necesitan un puntero y evitar que se desalojen páginas compartidas al finalizar un proceso (lo cual requiere estructuras especiales).

Cuando en un momento un proceso intenta escribir se produce un TRAP y se crea una copia donde se escriben los datos que quiera el proceso ofensor. Ahora ambas se establecen como *LECTURA/ESCRITURA*.

A este proceso se le llama **Copiar en Escritura**.

Bibliotecas compartidas

Son bibliotecas extensas que se utilizan por muchos procesos. Se pueden establecer dos enlaces:

- **Enlace Estático**: las funciones se establecen en el espacio de direcciones de cada proceso. Sólo se cargan las páginas que se utilizan. --> **utiliza mucha memoria con páginas repetidas**.
- **Enlace Dinámico (DLL)**: Tiene una rutina que enlaza la función al proceso llamador en tiempo de ejecución.

Archivos asociados a memoria

Un proceso puede asociar un archivo a una porción de su espacio de direcciones virtuales; se cargan las páginas según su demanda.

Varios procesos pueden asociar el mismo archivo. Es lo que se llama **Memoria Compartida**, que puede ser un canal de comunicación entre procesos (Ej: un .txt en el que van escribiendo dos procesos).

Política de Limpieza

Cuando se pide un marco de página libre y no hay, es necesario pedir una página y escribir la anterior en el disco.

Para eso existen los **Demonios de Paginación**, está la mayor parte del tiempo inactivo pero se despierta de forma periódica para liberar páginas mediante el algoritmo seleccionado.

```
*NOTA: También se asegura que no necesita escribir en disco cuando se requiere
un marco de página.
```

Cuestiones de Implementación

NO RESUMIDAS

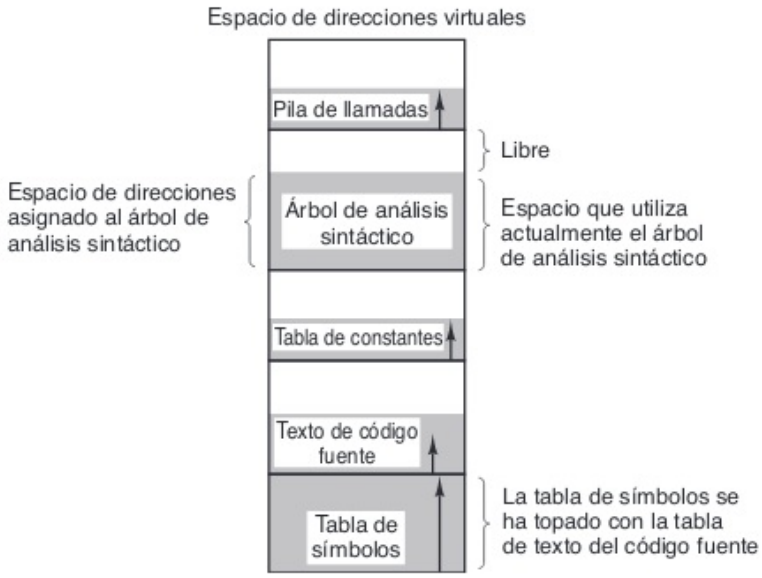
Segmentación

La memoria virtual analizada hasta ahora era unidimensional. SIn embargo puede haber casos en los que sería

mejor tener más de una.

```
Por ejemplo, un Compilador necesita tener su tabla de constantes, su pila, el
texto del código...
```

Algunas de esas tablas crecen durante la compilación, otras crecen y disminuyen (pila). En una memoria unidimensional, se deberían usar trozos contiguos como en la figura.



Si suponemos que puede existir algún programa muy extenso que puede llenar alguna de las secciones se puede considerar varias formas... desde interrumpir la compilación, reasignar el espacio quitándoselo a secciones poco llenas u alguna otra.

Sin embargo, la **solución** es realmente liberar al programador de administrar sus tablas.

Esta solucion consiste en proporcionar muchos espacios de direcciones independientes llamados **segmentos** (cuyo valor va de **0** a **x**). De esta forma los segmentos pueden expandirse y contraerse sin afectar unos a otros.

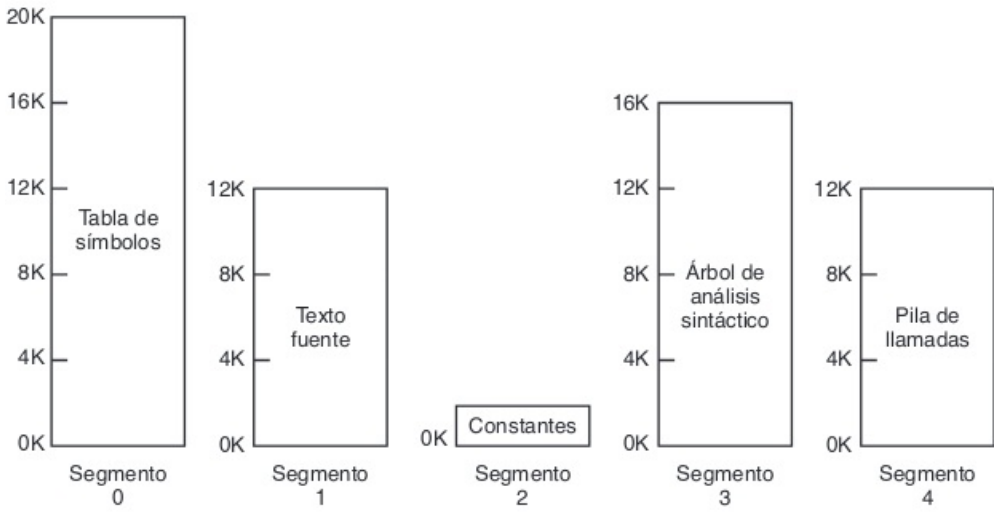


Figura 3-32. Una memoria segmentada permite que cada tabla crezca o se reduzca de manera independiente a las otras tablas.

Hay que considerar que otra ventaja es la **facilidad de compartir bibliotecas, procedimientos...**. Ya que las bibliotecas se pueden colocar en un segmento y varios procesos pueden compartirla, eliminando la necesidad de tenerla en el espacio de direcciones.

Implementación de segmentación pura.

Consideración	**Páginas**	**Segmentación**
Necesita el programador saber qué técnica se usa?	No	Sí
Cuantos espacios de direcciones lineales hay?	1	Muchos
Puede exceder el espacio de direcciones virtual el tamaño total de RAM?	Sí	Sí
Pueden tener distinta protección los datos y procedimientos?	No	Sí
Pueden las tablas fluctuar el tamaño con facilidad?	No	Sí
Se facilita la compartición de procedimientos entre usuarios?	No	Sí

- **Paginación**: Se inventó para obtener un gran espacio de direcciones lineal sin tener que comprar más memoria física.
- **Segmentación**: Se inventó para permitir a los programas y datos dividirse en espacios de direcciones lógicamente independientes, ayudando a la compartición y protección.

Segmentación con Paginación

Si los segmentos son extensos es recomendable paginarlos y combinar los dos métodos. Se necesita una **tabla de segmentos** por proceso y para cada segmento su **tabla de páginas**.