SISTEMAS OPERATIVOS 2016

Resumen del libro Dahlin

2 LA ABSTRACCIÓN DEL KERNEL

2.1 El proceso de abstracción

Un *proceso* es una instancia de un *programa*. Cada programa puede tener 0, 1 o más procesos ejecutándose. Para cada instancia de un programa, hay un proceso con su propia copia del programa en memoria.

El SO sigue los pasos de varios procesos usando una estructura llamada *bloque de control de procesos (PCB: Process Control Block)*. Este bloque tiene todo sobre los procesos: donde está alojado en memoria, donde está su imagen ejecutable en el disco, etc.

Cada proceso tiene su program counter, código, datos, heap y pila. Algunos programas tienen *múltiples actividades o hilos*, donde cada una de estas tiene su propio PC y pila, pero todas operan con el mismo código y datos que los otros hilos. Un SO corre múltiples hilos por proceso.

Procesos, procesos ligeros e hilos

Un proceso era antes lo que ahora es un *hilo*: una secuencia lógica de instrucciones que se ejecuta en el SO o en una aplicación.

Antes, los programas de usuarios eran de un solo hilo, con un solo stack y un solo PC. No había confusión.

Cuando las computadoras en paralelo se empezaron a popularizar, cambió todo. Un programa en un multiprocesador puede correr varias secuencias de instrucciones *en paralelo*. Cada una con su propio PC, pero todas cooperando bajo el mismo límite de protección. Estas secuencias comenzaron a llamarse *procesos ligeros*, sin embargo el término *hilo* se hizo más popular.

Un proceso ejecuta un programa, que consiste en uno o más hilos corriendo dentro de un límite de protección.

2.2 Operación en modo dual (Dual-Mode)

Para evitar que algunos procesos afecten al SO, en el hardware se implementan dos modos: *kernel* y *usuario*. Ambos son representados por un bit en el registro de estado del procesador.

- *Modo usuario:* el procesador chequea cada instrucción antes de ejecutarla, para verificar que el proceso pueda ejecutarla.
- *Modo kernel:* se ejecuta todo sin chequeo.

Cualquier error en el kernel puede traer errores en el SO.

Separando el código que no necesito que esté en el kernel, el SO puede ser más fiable. Esto ilustra el *principio del mínimo privilegio*, que asegura que la seguridad y confianza del SO están aseguradas si cada parte del SO tiene exactamente los privilegios que necesita y nada más.

Lo que debe hacer el hardware para evitar que las aplicaciones del kernel y del usuario se afecten entre sí es:

- *Instrucciones privilegiadas:* todas las instrucciones inseguras son prohibidas cuando se ejecutan en modo usuario.
- *Protección de memoria:* todos los accesos de memoria fuera del rango válido de un proceso son prohibidos en modo usuario.
- *Interrupciones temporales:* independiente de lo que haga el proceso, el kernel debe poder tomar el control del proceso de forma periódica.

Instrucciones privilegiadas

Un proceso puede cambiar su nivel de privilegio indirectamente ejecutando una llamada al sistema. Las instrucciones privilegiadas son aquellas que se encuentran en modo kernel pero no en modo user. El SO debe ejecutar éstas para hacer bien su trabajo: cambiar privilegios, ajustar accesos de memoria, etc. No es conveniente que estén en modo user porque afectarían a todo el sistema. Una de las primeras rutinas que hace el kernel es verificar los privilegios que tiene una aplicación, y ver si se puede ejecutar o no.

Protección de memoria

Para correr un proceso, tanto el SO como la aplicación, deben estar en memoria al mismo tiempo. Para hacer la memoria segura, el SO debe configurar el hardware para que cada proceso de aplicación lea y escriba su propia memoria, no la del SO, de esta forma evita que las aplicaciones influyan en el sistema. Un ejemplo de esto sucedía en MS-DOS, donde esto no estaba contemplado, y los bugs de una aplicación hacían que se caiga el sistema.

¿Cómo hace el SO para que el user no acceda a memoria física?

Para que esto no suceda, el procesador tiene dos registros llamados *base & bound (base y límite)*, ubicados en el kernel, donde el user no puede alterar sus valores. La *base* marca el inicio de la región de memoria del proceso a memoria física y *bound* el final.

Cada vez que el procesador ejecuta una instrucción, chequea si el program counter está dentro del base bound, y si lo está, procede con la ejecución.

El kernel no respeta el base bound, pudiendo acceder a cualquier parte de la memoria.

Las desventajas del base bound son:

- *Heap y pila expandibles:* no se puede expandir la porción de memoria que se le da a un programa. Dependiendo del comportamiento del programa, este se puede expandir o no.
- *Memoria compartida:* el base bound no permite compartir memoria entre procesos.
- *Direcciones de memoria física:* cada vez que el programa es cargado en memoria, el kernel debe cambiar cada ubicación de instrucción o datos que haga referencia a una dirección global.
- *Fragmentación de memoria:* los programas se albergan en memoria de forma fragmentada, esto sucede a tal nivel que luego no hay lugar contiguo para otros procesos, a pesar de que haya memoria disponible.

Para arreglar los problemas mencionados, los procesadores crean *direcciones virtuales*. Los procesos creen que estas memorias son “toda la máquina” y las ocupan, luego el hardware las traduce en memoria física.

Interrupciones de tiempo

Cuando un programa se ejecuta a nivel usuario, éste piensa que tiene todo el poder sobre el CPU, sin embargo, esto es una simple ilusión, ya que el SO es quien tiene control del CPU, y también del programa ejecutado. Puede detener la aplicación siempre que lo desee.

2.3 Tipos de modo de transferencia

Existen 3 razones para que el kernel tome el control por sobre el usuario: interrupciones, excepciones de procesador y llamadas al sistema.

Modo usuario a modo kernel

- *Interrupciones:* señal asincrónica que le indica al procesador que un evento externo ha ocurrido y necesita atención. El procesador ejecuta instrucciones y va chequeando por interrupciones. Si encuentra una, termina la instrucción, atiende la interrupción, habiendo guardado antes el estado de la instrucción, y luego continúa. Una alternativa es el *polling*, donde el kernel va realizando un loop chequeando cada dispositivo de E/S y viendo si hay algún evento que deba atender. En los multiprocesadores, un proceso puede enviar una interrupción a otro proceso.

- *Excepciones de procesador:* es un evento de hardware causado por un programa a nivel usuario, cuyo comportamiento causa una transferencia del control al kernel. Al igual que con una interrupción, el hardware termina las instrucciones previas, guarda el estado y ejecuta el *manipulador (handler) de excepciones* en el kernel.

- *Llamadas al sistema:* es cualquier procedimiento proveído por el kernel que puede ser llamado desde el nivel usuario. Al igual que los anteriores, cambia el estado a modo kernel. Un SO puede proveer cualquier número de llamadas al sistema. Cuando el kernel completa la llamada, vuelve al modo user para continuar ejecutando las instrucciones.

Modo kernel a modo usuario

- *Nuevo proceso:* para crear un proceso nuevo, el kernel copia el programa en memoria, setea el PC, setea el puntero de pila y cambia a modo user.
- *Retomar después de interrupción/excepción/llamada:* cuando el kernel termina su respuesta, regresa a la ejecución del proceso interrumpido, restaurando el PC y los registros, y volviendo a modo user.
- *Cambiar a diferentes procesos:* luego de atender la interrupción, excepción o llamada, puede volver a otro proceso, no exclusivamente al interrumpido. Luego vuelve a modo usuario.
- *User-level upcall:* hay SO que soportan programas de usuario con la habilidad de recibir notificaciones asincrónicas de eventos, similares a lo del kernel pero en modo usuario.

2.4 Implementación del modo de transferencia

Vector de interrupciones

Cuando ocurre una interrupción, excepción o llamada, el SO debe tomar distintas acciones dependiendo del tipo de evento que se trate. ¿Cómo sabe esto el SO? ¿Qué evento tomar?

El procesador tiene un registro especial apuntando a una parte de la memoria llamada *vector de interrupciones*, que es un arreglo de punteros donde cada puntero apunta a la primera instrucción de un handler en particular.

Formato del array: 0-31 para los diferentes tipos de excepciones, 32-255 para los diferentes tipos de interrupciones, y 64 lugares para llamadas al sistema. El hardware determina que evento surgió y llama al handler correspondiente.

Pila de interrupciones

En la mayoría de los procesadores hay un registro que apunta a cierto lugar de la memoria llamado *pila de interrupciones*. Es donde se guardan los estados de los procesos interrumpidos.

La interrupción ocurre, entonces antes de llamar al handler, se pushean los registros del proceso en la pila. También el hardware pone al puntero de pila del proceso apuntando a la pila del kernel.

Se necesita una pila en kernel y una en user por dos razones:

- *Confianza:* la pila a nivel user puede tener accesos a memoria no válidos.
- *Seguridad:* en multiprocesos, los hilos pueden correr al mismo tiempo y modificar la memoria durante la llamada al sistema.

En multiprocesador, cada procesador necesita su pila de interrupciones.

Estados de los procesos y las pilas

• Si el proceso está corriendo en modo user, la pila del kernel está vacía esperando una interrupción.

• Si el proceso está corriendo en modo kernel, se usa la pila kernel para guardar los datos del proceso.

• Si el proceso está listo para correr pero está esperando al procesador, la pila kernel tiene los registros y el estado del proceso para cuando retorne.

• Si el proceso está esperando que se termine un evento de E/S, la pila kernel tiene la computación suspendida para cuando el evento de E/S finalice.

Enmascaramiento de interrupciones

Cuando vienen muchas interrupciones a la vez, se usa la instrucción *disable interrupts*, que las enmascara, no las borra, para atenderlas más tarde. Para activarlas nuevamente se usa *enable interrupts*. Esto se hace a nivel kernel.

Como el hardware es limitado, tiene un buffer limitado para las interrupciones pendientes, por lo tanto, algunas se pueden perder si están en modo disable por mucho tiempo. El handler es el encargado de ver si hay eventos de E/S pendientes.

2.5 Modo de transferencia x86

Pasos que provocan el inicio del handler (EJERCICIO PSEUDOCODIGO LLAMADA):

1) Enmascara interrupciones.
2) Guarda 3 valores clave: los valores del puntero de pila, banderas de ejecución y el puntero de instrucciones en registros temporales.
3) Apunta el puntero de la pila del proceso a la pila kernel.
4) Inserta los 3 valores en la pila.
5) Genera código de error (opcional).
6) Se invoca al handler, obteniendo el índice en el vector de interrupciones.

2.7 Iniciar un nuevo proceso

Para arrancar un nuevo proceso a nivel usuario, el kernel debe:

- Ubicar e inicializar la *tabla de control de procesos (PCB).*
- Ubicar memoria para el proceso.
- Copiar el programa del disco al nuevo lugar de memoria.
- Ubicar una pila de nivel usuario para la ejecución.
- Ubicar una pila kernel para administrar interrupciones.

Además debe:

- Copiar argumentos en memoria del user.
- Transferir el control a modo user.

3 LA INTERFAZ DE PROGRAMACIÓN

3.1 Gestión de procesos

Manejo de procesos en Windows

Para crear un proceso usamos *CreateProcess:* boolean CreateProcess(char *prog, char *args). Los pasos que realiza el kernel para el CreateProcess son:

- Crea e inicializa la tabla de procesos en el kernel (PCB).
- Crea e inicializa un nuevo espacio de direcciones.
- Carga el programa *prog* en el espacio de direcciones.
- Copia los argumentos *args* en la memoria del espacio de direcciones.
- Inicializa el hardware para que se ejecute en "*start*".
- Informa al planificador que el proceso está listo.

Manejo de procesos en Linux

Divide el CreateProcess en dos pasos llamados *fork* y *excec*:

- *Fork:* crea una copia exacta del proceso padre (vamos a tener que necesitar una forma de distinguir padre e hijo). Una vez que el contexto esté seteado, el hijo llama a *excec*. Excec trae una imagen ejecutable en memoria y la ejecuta. El fork no recibe argumentos y devuelve un entero.
- *Excec:* recibe dos parámetros, el nombre del programa a correr y los argumentos en forma de arreglo. Esto es en lugar de los 10 parámetros que hay que pasarle a CreateProcess. El excec n0o crea un proceso (excec does not create a process!).

Los pasos para implementar el fork son:

1) Crear e inicializar la tabla de control de procesos (PCB).
2) Crear un nuevo espacio de direcciones.
3) Inicializar un nuevo espacio de direcciones con una copia de todo el contenido del espacio de direcciones del padre.
4) Heredar el contexto de ejecución del padre. Por ejemplo, todos los archivos abiertos.
5) Informar al planificador que el proceso está listo.

Para el hijo, fork() devuelve cero, para el padre devuelve el ID del hijo. La llamada excec es la que completa los pasos para correr un nuevo programa.

Los pasos del excec son:

1) Carga el programa *prog* en el espacio de direcciones.
2) Copia los argumentos *args* en la memoria de dicho espacio.
3) Inicializa el hardware para que comience la ejecución en "*start"*.

El *wait* pausea al padre hasta que el hijo termine, se bloquee o este termine.

4 CONCURRENCIA E HILOS

4.1 Casos de uso de hilos

En un programa, podemos representar cada tarea concurrente como un *hilo*. Cada hilo provee una abstracción de una secuencia de ejecución similar al modelo de programación básico. Podemos pensar en estos programas clásicos como *programas de un hilo*, con sólo una secuencia lógica de pasos, donde cada instrucción acontece a otra.

Un programa *multi-hilo* es una generalización del mismo modelo de programación, donde cada hilo sigue una secuencia de instrucciones. Sin embargo ahora un programa puede ejecutar varios hilos al mismo tiempo.

Razones para usar hilos

- *Expresar tareas concurrentes:* los programas interactúan o simulan aplicaciones del mundo real que tienen actividades concurrentes. Los hilos nos permiten justamente eso, expresarlas mediante diferentes hilos.
- *Trabajo de fondo (background):* para mejorar la responsabilidad del usuario y el rendimiento, una estrategia posible es la de crear hilos que operen en segundo plano (background), para evitar tener al usuario esperando por un resultado. Para mantener la responsabilidad de la interface, podemos usar hilos para dividir el comando en dos partes: cualquier cosa que pueda ser ejecutado en el main y un hilo separado que se encargue de las actividades del background.
- *Explotar multiprocesadores:* los programas pueden usar hilos en un entorno multiprocesador para lograr *paralelismo*, y así acelerar considerablemente el rendimiento. Una ventaja de usar los hilos para paralelismo, es que el número de hilos no necesariamente debe ser igual al número de procesadores físicos. El sistema operativo se encarga de ir alternándolos.
- *Manejo de dispositivos de E/S:* para lograr un buen trabajo, las PC's deben interactuar con el mundo vía dispositivos de E/S. Corriendo tareas en hilos separados, cuando una tarea está esperando por el dispositivo, el procesador puede progresar en una tarea diferente. Esto es beneficioso por dos lados:
    1) El procesador es mucho más rápido que los sistemas de E/S, por lo tanto sería un desperdicio no usarlo mientras esperamos por un dispositivo.
    2) Los eventos de E/S son impredecibles, por lo tanto, el procesador debe ser capaz de trabajar sobre otras tareas mientras sigue respondiendo de forma eficaz a los dispositivos de E/S.

4.2 Abstracción de hilo

Viendo la definición de hilo tenemos que: *"un hilo es una secuencia de instrucciones individual que representa una tarea separadamente planificable".*

Desmenuzando:

- *Secuencia de instrucciones individual:* sigue el modelo secuencial de programación.
- *Tarea separadamente planificable:* el SO puede correrla, suspenderla o reanudarla cuando quiera.

Correr, suspender y reanudar hilos

Los hilos dan la ilusión de un infinito número de procesadores. Para esto, el SO ejecuta instrucciones de cada hilo para que cada uno tenga progreso. Para lograr esto, el SO usa un *planificador de hilos* para ir cambiando entre los hilos que están corriendo y aquellos que están listos (pero no corriendo).

El cambio de hilos es transparente al código. La abstracción hace parecer que cada hilo es siempre una cadena de instrucciones. Esto significa que el programador ve sólo el código que quiere ver, sin rastros de otros hilos.

Por ejemplo:

Visión del programador	***Posible ejecución***	***Otra ejecución***
⋮	⋮	⋮
$x = x + 1;$	$x = x + 1;$	$x = x + 1;$
$y = y + x;$	$x = y + x;$	Hilo suspendido
$z = x + 5y;$	$z = x + 5y;$	Otro hilo corriendo
⋮	⋮	Retomamos el hilo
⋮	⋮	$y = y + x;$
⋮	⋮	$z = x + 5y;$

El programador siempre ve sólo las ecuaciones (se ejecuten hilos o no).

4.3 Funciones de hilos simples

- void thread_create(thread, func, arg): crea un nuevo hilo. Concurrentemente con la llamada al hilo, ejecuta la función *func*, con los argumentos *arg*.
- void thread_yield(): libera el procesador para que lo ocupe otro hilo.
- int thread_join(thread): espera a que *thread* termine (si aún no lo ha hecho). Luego devuelve el valor que ese hilo le pasó a *thread_exit*. Sólo puede ser llamado una vez por hilo.
- void thread_exit(ret): termina el hilo y guarda en *ret* la estructura del hilo.

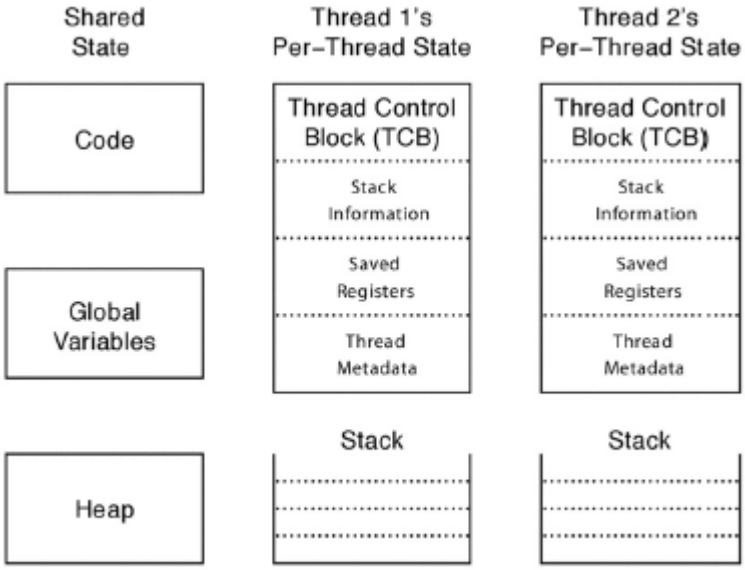
Estas funciones nos permiten invocar *llamadas asincrónicas de procedimiento*. Una llamada de procedimiento normal pasa argumentos a una función, la corre en la pila del *llamador* y cuando finaliza devuelve el control al llamador con el resultado.

La asincrónica separa la llamada del resultado (*return*): con *thread_create*, el llamador llama a la función, pero a diferencia de la llamada normal, el llamador continúa su ejecución concurrentemente con la función llamada. Luego, el llamador puede esperar a que termine la función.

El *thread_create* es análogo al *fork* y al *excec*, y el *thread_join* es análogo al *wait*.

4.4 Estructuras de datos de hilos y ciclo de vida

El SO provee la ilusión de que cada hilo tiene su propio procesador virtual, mediante la suspensión y retomación de hilos. Para que esto funcione, el SO debe guardar y restaurar los estados de los hilos. Sin embargo, como los hilos corren tanto en modo user como kernel, hay *estados compartidos* que no se guardan o restauran cuando se van cambiando los hilos.



El bloque de control almacena el estado del hilo: el estado presente (los registros y un puntero a pila) y los metadatos necesarios para el manejo del hilo. Estado compartido tiene lo que se muestra.

El SO necesita una estructura que represente el estado de un hilo. Para ello utiliza el *bloque de control de hilos (TCB: thread control block)*. Por cada hilo el SO crea un bloque.

El TCB tiene dos tipos de información por hilo:

- El *estado de ejecución* que realiza el hilo.
- *Metadatos* sobre el hilo.

Estado por hilo

Para crear múltiples hilos y manejarlos como quiera, el SO debe ubicar espacio en el TCB para el estado actual de cada ejecución del hilo: necesita un *puntero a la pila* del hilo y una *copia de los registros*:

- *Pila:* guarda información necesaria sobre los procedimientos anidados que el hilo está ejecutando. Tiene un *stack frame* por cada procedimiento anidado. Cada hilo necesita su propia pila y al crearla también se crea un puntero en el TCB.
- *Copia de registros del procesador:* no sólo los registros generales, sino los registros también de instrucción (*instruction pointer*) y el de pila (*stack pointer*).

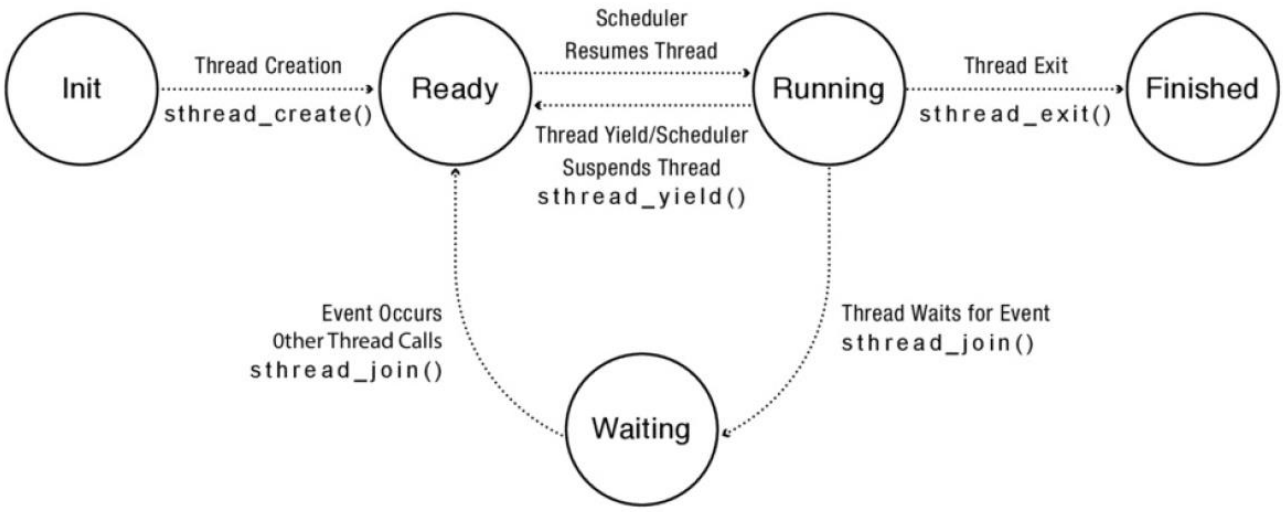
Para correr o suspender los hilos, el SO debe guardar los registros del hilo que no se esté usando. En algunos sistemas los registros generales son guardados en el tope de la pila y el TCB sólo tiene un puntero. En otros SO están todos los registros en el TCB.

En los *metadatos por hilo* simplemente se tiene información para manejar el hilo.

Estado compartido

Opuesto al estado por hilo, algunos estados son compartidos entre los hilos que están corriendo (en el procesador o en el SO). El código es compartido por todos los hilos de un proceso, aunque pueden ejecutar cada uno distintas partes del código. Las *variables globales* son almacenadas de forma estática y el *heap* de forma dinámica.

4.5 Ciclo de vida de un hilo ☺



• *Init:* coloca al hilo en estado *init* e inicializa las estructuras de datos por hilo. Una vez realizado pasa al estado *ready (listo)*, agregando el hilo a la *ready list (lista de hilos listos)*. Esta lista es un conjunto de hilos que están listos para correr pero que esperan su turno en el procesador. La lista no es literalmente una lista, sino que se usa una *cola de prioridad*.

• *Ready:* un hilo en estado *ready* está listo para correr pero no lo hace en ese instante. Su TCB está en la *lista de hilos listos* y los valores de sus registros están en su TBC. En cualquier momento el planificador puede hacer pasar un hilo del estado *listo* a running (*corriendo)*, esto lo hace copiando los valores de registros que estaban en su TCB a los registros del procesador.

• *Running:* el hilo corre en el procesador. En este momento los valores de los registros del hilo están en el procesador. Un hilo puede pasar de *corriendo* a *listo* de dos formas:

1) Primera forma: guarda los registros en el TCB y luego cambia para que el procesador corra el próximo hilo en la *ready list*. Esto lo hace el planificador.
2) Segunda forma: un hilo voluntariamente puede desistir de usar el procesador (no tiene aguante) y pasar de *corriendo* a *listo*. Esto se lleva a cabo con *thread_yield*.

Notar que un hilo puede ir de *listo* a *corriendo* múltiples veces. Desde que el SO guarda y restaura los registros del hilo exactamente sólo la velocidad de la ejecución del hilo es afectada en esto. De más está decir que un hilo en *running* no está en la *lista de hilos listos*.

• *Waiting:* un hilo en *waiting* espera algún evento. Un hilo en *waiting* no puede ejecutar una acción hasta que otro hilo lo mueva a *listo*. Mientras espera un evento no puede progresar. El TCB está alojado en la *waiting list (lista de espera)*, asociada a alguna variable de sincronización asociada con el evento. Cuando el evento ocurre, el SO mueve el TCB de la *waiting list* a la *ready list* del planificador.

• *Finished:* un hilo en este estado *no corre más*. Por un tiempo limitado, algunos datos se mantienen por si otro hilo necesita los mismos. Esto se encuentra en la *finished list (lista finalizada)*.

Estado	Localización del TCB	Localización de Registros
INIT	Se crea	TCB
READY	Ready List	TCB
RUNNING	Running List	Procesador
WAITING	Waiting List	TCB
FINISHED	Finished List o Borrado	TCB o Borrado

5 SINCRONIZACIÓN DE ACCESO A LOS OBJETOS COMPARTIDOS

5.1 Desafíos

Condiciones de carrera

Una *condición de carrera* ocurre cuando el comportamiento de un programa depende de la intercalación de operaciones de diferentes hilos. "Los hilos corren una carrera entre sus operaciones y la ejecución del programa depende de quien gane esa carrera".

Por ejemplo:

Suponga que $y = 12$, donde tenemos dos hilos:

Hilo (1): $x = y + 1$.

Hilo (2): $y = y * 2$.

Si (1) se ejecuta primero (gana la carrera) y luego se ejecuta el otro, la salida será $x = 13$.

Si (2) se ejecuta primero y luego se ejecuta el otro, la salida será $x = 25$.

Concepto de operaciones atómicas

Son operaciones que no pueden ser intercambiadas o divididas por otras operaciones. Los ejemplos anteriores son operaciones atómicas.

Los programas donde hay riesgo de condición de carrera deben cumplir o intentar cumplir dos principios:

- *Seguridad:* el programa no entra nunca en un *mal estado*.
- *Viveza (liveness):* el programa eventualmente entra en un *estado bueno*.

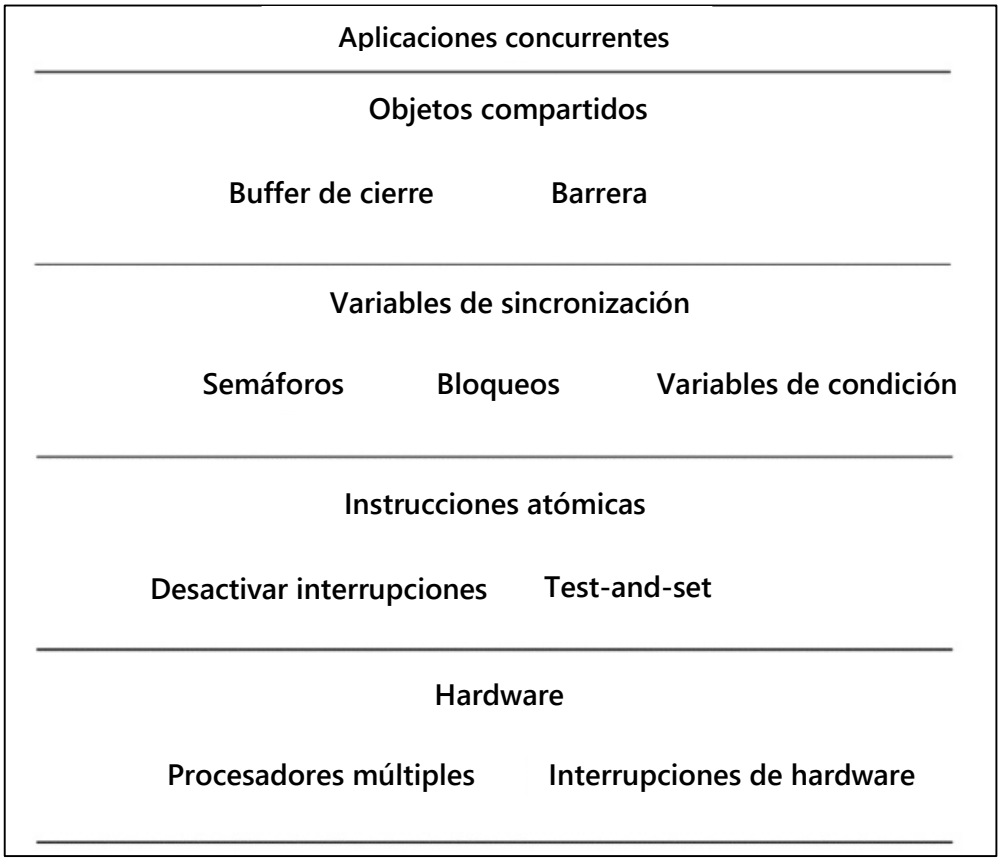
5.2 Objetos compartidos

Los objetos compartidos son objetos que pueden ser accedidos de forma segura por múltiples hilos. Todos los estados compartidos en un programa deben ser encapsulados en uno o más objetos compartidos.

Los objetos compartidos ocultan los detalles de sincronización de acciones de múltiples hilos detrás de una *interfaz limpia*. Cada clase de objetos compartidos definen un conjunto de métodos públicos en los cuales los hilos operan.

Implementación

Internamente los objetos compartidos deben manejar los detalles de sincronización. Estos objetos se implementan en capas:



Los programas multihilos son construidos con operaciones compartidas. Las operaciones compartidas son construidas usando variables de sincronización y variables de estado. Las variables de sincronización son implementadas usando instrucciones especiales del procesador para manejar el envío de interrupciones y para leer/escribir lugares de memoria atómicos.

Capas:

- *Capas de objetos compartidos:* definen una lógica de aplicación específica y esconde los detalles de implementación interna.
- *Capa de variables de sincronización:* una variable de sincronización es una estructura de datos usada para coordinar accesos concurrentes a estados compartidos. Construimos las operaciones compartidas usando dos tipos de sincronización de variables: *locks (bloqueos)* y *variables de condición*. Las variables de sincronización coordinan el acceso a variables de estado.
- *Capa de instrucciones atómicas:* internamente, variables de sincronización deben manejar las intercalaciones de acciones de distintos hilos.

5.3 Locks (bloqueo)

Es una variable de sincronización que provee exclusión mutua. Cuando un hilo tiene un lock, ningún otro hilo puede tenerlo, es decir, los otros hilos están excluidos. Estos locks provocan que el acceso a los estados compartidos sean de a un hilo.

Un lock provee exclusión múltiple por dos métodos: `lock::acquire()` y `lock::release()`. Estos métodos siguen los siguientes principios:

- Un lock puede estar *libre* u *ocupado*.
- Un lock está en estado *libre* al inicio.
- *lock::acquire* espera a que el lock esté *libre* y lo ocupa. Éstas son acciones atómicas.
- *lock::release* lo hace *libre*. Si hay *acquires* pendientes ocupa el próximo.

Un lock debe asegurar las siguientes tres propiedades:

- *Exclusión mutua:* como mucho un hilo tiene el lock.
- *Progreso:* si ningún hilo tiene lock y algún hilo trata de obtenerlo, en algún momento consigue dicho lock.
- *Espera encerrada:* si el hilo T trata de adquirir el lock, entonces existe un número limitado de veces para que otro hilo pueda adquirirlo antes de que T lo haga.

La *exclusión mutua* es una propiedad segura porque evita que más de un hilo acceda a los estados compartidos. En cambio, *progreso* y *espera* son propiedades de viveza. Si un lock está libre, eventualmente un hilo puede adquirirlo.

Secciones críticas

Una sección crítica es una secuencia de código que accede de forma atómica a los estados compartidos. Se deben tener en cuenta dos cosas:

- Cada clase puede definir múltiples métodos para operar en estados compartidos. Por ende, puede haber muchas secciones críticas por clase. Sin embargo, para cada instancia de clase, un solo hilo tiene el lock, por ende, una sola sección crítica es ejecutada.
- Un programa puede crear múltiples instancias de clase. Cada instancia es un objeto compartido, cada uno con un lock. Además, diferentes hilos pueden estar activos en las secciones críticas para diferentes instancias de objetos compartidos.

5.8 Semáforos

Son utilizados para sincronización en el sistema operativo. Se definen como sigue:

- Un semáforo tiene un valor no negativo.
- Cuando se lo crea, el valor puede ser inicializado con cualquier número distinto de cero.
- *Semaphore::P()* espera a que el valor sea positivo. Cuando lo consigue, lo decrementa por uno y devuelve.
- *Semaphore::V()* automáticamente incrementa el valor en 1.

Ninguna otra operación está permitida en un semáforo. Ningún hilo puede leer directamente el valor actual del semáforo.

La operación P es una operación atómica. Como resultado, un semáforo nunca puede tener valor negativo, incluso cuando múltiples hilos llaman a P a la vez. Con ésta definición, los semáforos pueden ser usados tanto para exclusión mutua (como lock) o esperar a un hilo para que haga algo.

Para usarlo como exclusión mutua, se inicializa en 1. Entonces, *semaphore::P()* es equivalente a *lock::acquire()*, y *semaphore::V()* es equivalente a *lock::release()*.

El libro plantea que es mejor usar locks y variables de condición que usar semáforos. Razones:

- Locks y variables de condición producen un código mejor documentado y más fácil de leer.
- Locks y variables de condición son mejor abstracciones para la espera generalizada que los semáforos.

7 SCHEDULING

Scheduling significa *planificación*. Diferentes políticas:

- ✓ FIFO (Primero en entrar, primero en salir).
- ✓ SJF (Trabajo más corto primero).
- ✓ ROUND ROBIN

Antes que nada definimos algunos conceptos:

- *Carga de trabajo:* es un conjunto de tareas que debe realizar un sistema. Define la entrada a un algoritmo de planificación.
- Algunas tareas son limitadas computacionalmente y otras limitadas a dispositivos de E/S. Las primeras sólo usan el procesador y las segundas gastan tiempo esperando dispositivos de E/S (pasan poco tiempo en la CPU).
- Un planificador debe ser *conservador (work-conserving)*, es decir, que nunca abandone el CPU si hay laburo.
- El planificador debe poder ceder el procesador a otra tarea. Esto ocurre si surge una interrupción de tiempo, o hay una tarea con mucha prioridad en la *ready list*.

7.1 FIFO

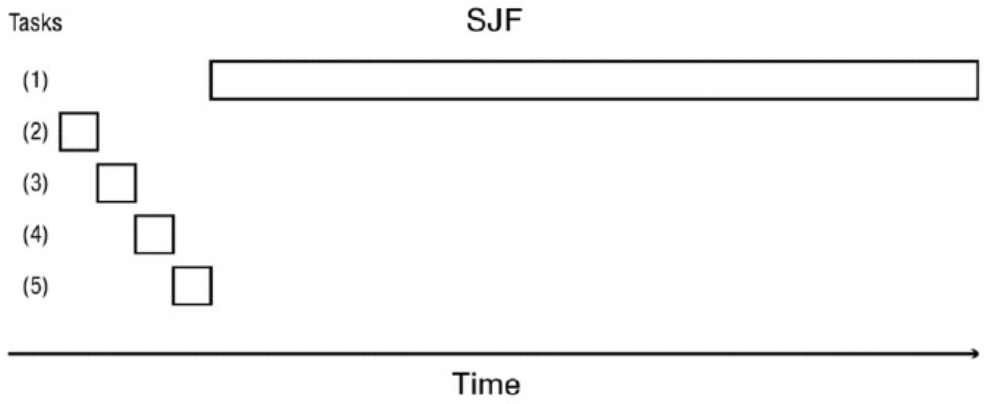
Primero en entrar, primero en salir. Es decir, la primera tarea que llega es la que se realiza. Minimiza la sobrecarga cambiando tareas sólo cuando finalizan. Definición de "*justo*". Cada tarea espera su turno y se realiza.

Desventaja: si una tarea pequeña está detrás de una que toma mucho tiempo, el sistema se vuelve ineficiente.

7.2 SJF

Para un tiempo de respuesta promedio, FIFO es malo. Para minimizar el tiempo usamos SJF: el trabajo pequeño primero.

Lo que cuenta como "*pequeño*", no es el tamaño original de la tarea, sino el tiempo que le queda a esa tarea. Por ejemplo, si estamos a un nanosegundo de terminar una tarea de una hora, nos quedamos con esa en lugar de elegir una tarea de un minuto que recién llega a la *ready list*.



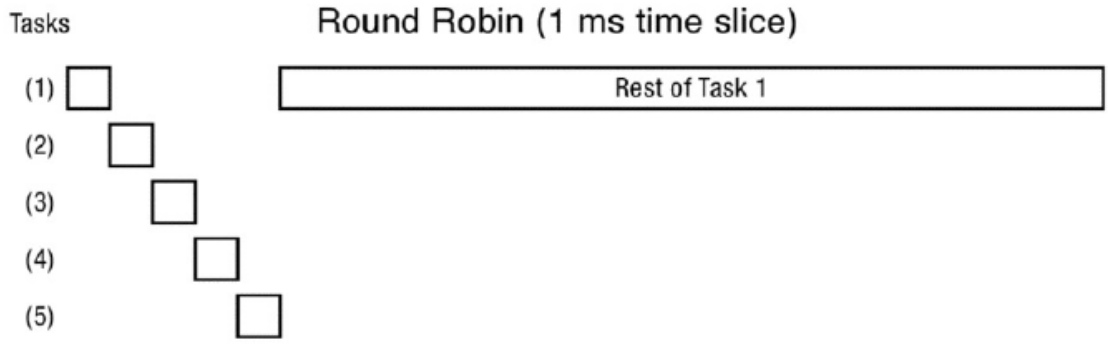
Desventajas:

- Imposible de realizar, ya que requiere conocer información del futuro.
- Pésima varianza temporal ya que las tareas largas las realiza de la forma más lenta posible.
- Si llegan muchas tareas pequeñas, las tareas grandes puede que nunca se completen. Si en la *ready list* hay tareas pequeñas y actualmente se están corriendo grandes, el planificador las intercambia.

7.3 ROUND ROBIN

Con RR el planificador toma la primera tarea de la *ready list* y le asigna una interrupción de tiempo para generar un delay, llamado *time quantum*. Cuando termina el time quantum, el procesador agarra la próxima tarea en la *ready list*. La tarea vuelve a la *ready list* para esperar su próximo turno.

Con RR no hay posibilidad de que una tarea se quede sin atender.



Nota: debemos elegir bien el *time quantum*. Un TQ muy chico provocará que el procesador cambie mucho de tareas y gaste más tiempo en eso que ejecutando las mismas tareas. Finalmente, si elegimos un TQ muy largo, las tareas tendrán que esperar mucho para su turno.

El libro compara al método con un estudiante estudiando para muchas materias, donde si lee sólo una hoja de un texto y cambia de materia para leer otra hoja, será poco efectivo. Por el contrario, si se concentra sólo en una materia, dejará las otras de lado.

Tomamos a RR como un punto medio entre FIFO y SJF. Por un lado si el TQ tiende a infinito, el RR se convierte en FIFO. Ahora, si asignaríamos un TQ por tarea (no uno global), se aproximaría a SJF (pero más lento).

Características generales:

- FIFO es simple y minimiza sobrecarga.
- Si las tareas difieren en tamaño, FIFO logra un tiempo promedio de respuesta malo.
- Si son de igual tamaño, FIFO tiene buen tiempo (óptima).
- SJF es pésima en términos de varianza temporal.
- Si las tareas son variantes en tamaño, RR aproxima a SJF.
- Si las tareas son iguales en tamaño, RR hace mal promedio de tiempo.
- Tareas que entremezclan procesador y E/S son beneficiosas con SJF y malas con RR.
- Usando colas con prioridad, de manera de hacer varios RR a la vez, podríamos alcanzar el balance entre sobrecarga y justicia (que cada tarea reciba la misma parte del procesador).

8 TRADUCCIÓN DE DIRECCIONES

8.1 Concepto de traducción de direcciones

En memoria tenemos un *traductor* que funciona como una caja negra, donde toma una instrucción y una memoria de referencia, chequea si es válida, y la convierte en memoria física.

Lo que pretendemos de ésta caja negra es:

- *Memoria protegida:* que un proceso acceda sólo a cierta parte de la memoria (prevenir el acceso a memoria ajena).
- *Memoria compartida:* que múltiples procesos compartan cierta región de la memoria.
- *Alojamiento de memoria flexible:* que el SO aloje procesos donde quiera dentro de la memoria física.
- *Direcciones escasas:* muchos programas tienen múltiples regiones de memoria dinámica, por lo que van cambiando de tamaño a lo largo de su ejecución. Los sistemas de 64 bits tienen direcciones expandibles para éste tipo de programas.
- *Tablas de transición compactas:* las estructuras de datos que usemos para traducir deben ser mínimas en comparación al tamaño de memoria física.
- *Portabilidad:* que el traductor sea compatible con distintas arquitecturas de procesador.

8.2 Traducción de direcciones flexibles

La caja negra de traducción consiste en dos registros extra por proceso. El registro *base* especifica el inicio de la región física de un proceso y el registro *límite* el final de esa región.

Ya que la memoria física puede contener varios procesos, el kernel resetea el contenido de *base bound* de acuerdo al proceso.

Realizar esto presenta ciertas desventajas:

- El programa podría sobre escribir su código, ya que opera con un proceso entero, no con sus hilos de forma independiente.
- También es difícil hacer regiones de memoria compartida entre dos procesos.
- Memoria dinámica difícil de implementar o imposible.

Segmentación

Soluciona las limitaciones del modelo anterior, brindando un arreglo de registros base/límite para cada proceso. Cada entrada en el arreglo controla una porción (o segmento) de una dirección virtual de memoria.

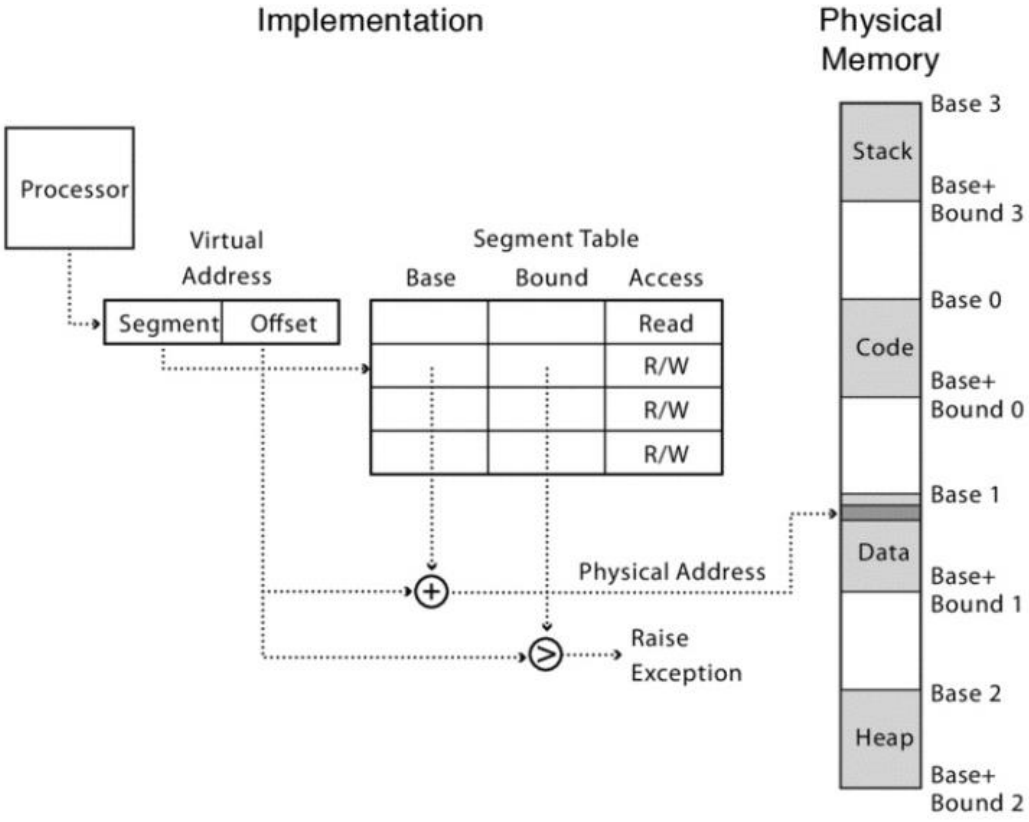
La memoria física para cada segmento es guardada de forma contigua, pero diferentes segmentos pueden ser guardados en diferentes lugares. El SO le puede dar distintos permisos a distintos segmentos. La memoria de un programa ya no es un contiguo sino un conjunto de regiones.

Si queremos acceder a un lugar donde la memoria tiene un hueco (tiene huecos debido a la segmentación) se produce una *falla de segmentación*.

Con segmentos, el SO puede permitir a procesos compartir algunas regiones de memoria y proteger otras. Finalmente mencionamos que tiene un buen manejo de la memoria alojada dinámicamente.

Desventajas:

- Sobrecarga por el manejo de una gran cantidad de variables y segmentos que crecen de forma dinámica.
- Debido a la segmentación, la memoria está tan dividida que tal vez haya lugar para un segmento pero no hay lugar contiguo. Esto se conoce como *fragmentación externa*.



Paginación

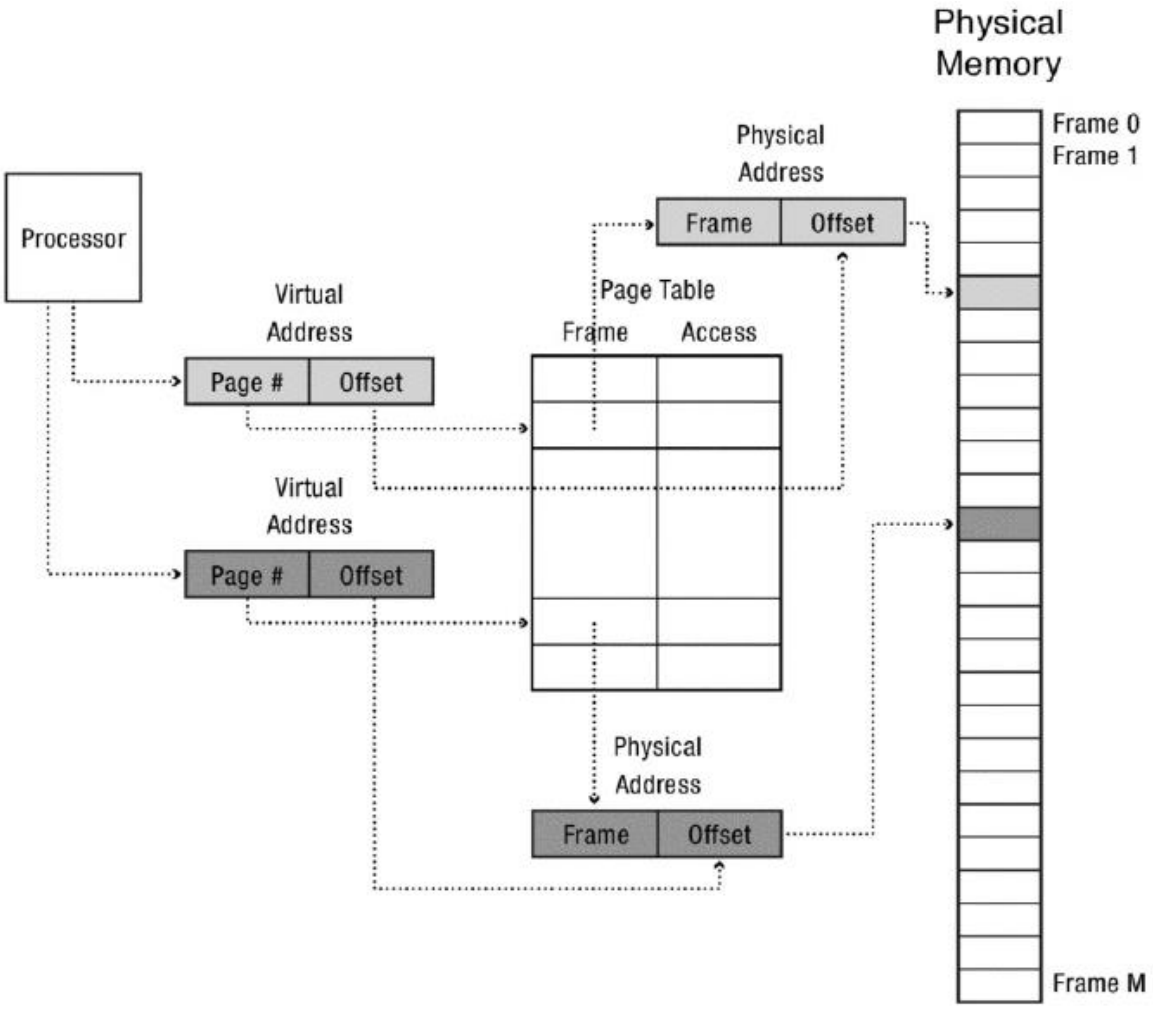
La memoria está alojada en celdas de tamaño fijo llamadas *page frame (marcos de página)*. La traducción se realiza de forma similar a la segmentación, pero en lugar de usar una tabla de segmento, donde las entradas tenían punteros a segmentos de tamaño variado, tenemos una *tabla de páginas* para cada proceso, donde cada entrada tiene punteros a *marcos de páginas.*

Cabe destacar que los marcos de página tienen un tamaño fijo que son potencias de dos.

No hay necesidad de un límite en el desplazamiento. La página entera se almacena en memoria como una unidad.

La paginación arregla una de las principales desventajas de la segmentación, que es la de encontrar un lugar libre en memoria.

El SO trata a la memoria como un mapa de bits, y ubicar un lugar libre es cuestión de ver esos bits.



Se puede reducir el tamaño de la tabla de páginas usando frames más grandes (celdas más grandes). Pero un tamaño grande de frame produce que un proceso ocupe poco lugar en la celda y desperdicie lugar. Esto se conoce como *fragmentación interna*.