Introducción a Sistemas Operativos

Sistema Operativo

Un ***sistema operativo*** es la capa de software que maneja los recursos de una computadora para sus usuarios y sus aplicaciones. Está diseñado para ocultar detalles de hardware de bajo nivel, y para crear una máquina abstracta que les proporcione a las aplicaciones servicios de alto nivel.

Interactúa directamente con el hardware, proporcionando servicios comunes a los programas.

Otra definición podría ser que puede definirse como el software que controla los recursos del hardware de una computadora, y proporciona un ambiente bajo el cual los programas pueden ejecutarse.

En un **sistema operativo** de propósito general, los **usuarios** interactúan con aplicaciones, estas aplicaciones se **ejecutan** en un **ambiente** que es proporcionado por el **sistema operativo**. A su vez el sistema operativo hace de mediador para tener acceso al hardware del equipo.

Time Sharing

La idea base del ***time sharing*** era transformar una computadora en un sistema multiusuario con tiempo de respuesta muy corto, de forma tal, que cada usuario creyera que estaba usando su propia computadora. En la actualidad, el concepto se refiere a compartir un recurso computacional de forma concurrente, con otros usuarios. Esto es para explotar al máximo la capacidad del CPU, mientras un usuario se encuentra inactivo o esperando resultados, pensando su siguiente movimiento, otros usuarios pueden ir realizando diferentes operaciones, ya que las computadoras son capaces de atender múltiples usuarios.

La forma principal de lograr la funcionalidad del SO es mediante la **virtualización**: el SO toma un recurso físico (memoria, el procesador, un disco) y lo transforma en algo virtual más general, fácil de usar. En otras palabras, utiliza el software para imitar las

características del hardware, y mostrar al usuario un sistema virtual.

Virtualización

La **virtualización** permite que muchos programas se ejecuten (compartiendo la CPU), que muchos programas accedan simultáneamente a sus propias instrucciones y datos (compartiendo la memoria) y que muchos programas accedan a los dispositivos (compartiendo así discos, etc.).

Referee

Debido a la virtualización, las aplicaciones comparten los mismos recursos físicos, y la memoria. El **referee** (OS) gestiona estos recursos compartidos, encargándose de decidir quién puede utilizar determinado recurso, y cuándo. Por ejemplo, un OS puede frenar la ejecución de una aplicación, e iniciar la ejecución de otra. Esto lo hace para protegerse a sí mismo, y a las demás aplicaciones que se encuentran ejecutando en la misma computadora.

Ilusionista

El sistema operativo proporciona a las aplicaciones una ilusión de que son capaces de utilizar toda la memoria, cuando en realidad no es así. Dicho en otras palabras, provee una abstracción del hardware, para simplificar el diseño de las aplicaciones, dejando a un lado detalles como por ejemplo: dónde se encuentra la memoria que se está utilizando, cómo esta se comparte con otras aplicaciones, etc.

Pegamento

El sistema operativo debe proveer servicios comunes que faciliten un mecanismo para interactuar entre las aplicaciones.

Kernel

El **SO** se ejecuta como la capa de software de más bajo nivel en la PC. Contiene una capa para la gestión de dispositivos, y una serie de servicios para la gestión de dispositivos agnósticos del hardware que son utilizados por las aplicaciones. En síntesis, estas dos capas se conocen como el **kernel** del SO.
Cuando se ejecuta código desde el kernel la PC pasa a un estado **modo supervisor**.
Las aplicaciones se ejecutan en un contexto aislado, protegido y restringido: **user**

mode, un modo más restrictivo, aislado y controlado.
El nombre de un archivo está dado por el path (camino) para localizarlo dentro de la estructura del sistema de archivos.

El **SO Unix** provee una lista de programas que vienen junto al SO, estos programas normalmente se denominan **comandos**, para realizar operaciones sobre los componentes del sistema de archivos: ls, pwd, cat, etc.

En Linux **todo es un archivo.**
Un programa es un archivo ejecutable, un **proceso** es una instancia de un programa en ejecución.
El **intérprete de comandos** o **shell** es el primer programa que se ejecuta (en unix) en modo usuario, esté a su vez ejecuta el comando login.

- **Shell** puede interpretar 3 tipos de comandos:
    - Comandos ejecutables, simples programas.
    - Shell scripts que es un archivo ejecutable.
    - Estructuras de control del shell if-then-else-if
- El **shell es un programa → no forma parte del Kernel.**
- El SO provee unas syscalls para la creación y manipulación de procesos: fork, exec, wait.

Bloques primitivos de construcción

.

- Unix provee ciertos mecanismos para que los programadores puedan construir a partir de pequeños programas otros más complejos.
- Estos mecanismos son usables desde el shell:
    - Redireccionamiento de entrada y salida de datos: **stdin, stdout y stderr.**
    - **>:** redirecciona el stdout.
    - **<:** redirecciona el stdin.
    - **Pipes:** permite que un stream de datos sean pasados entre dos procesos, uno escritor y el otro lector.

Kernel

Ejecución directa

El SO crea un punto de entrada en la lista de procesos, luego se reserva memoria para alojar el programa, se carga a memoria el mismo, se setea el stack, se limpian los registros y se ejecuta la llamada al main del programa, el cual retorna 0. Una vez finalizada la ejecución, se libera la memoria reservada para el programa, y se elimina la entrada en la lista de procesos.

Esta ejecución es muy veloz, pero tiene algunos problemas. Por ejemplo, ¿cuándo se corre un programa, como se asegura el SO de que el programa no va a hacer nada que el usuario no quiere que sea hecho? ¿Cómo hace el SO para pausar la ejecución de ese programa, y hacer que otro sea ejecutado?

En resumen, correr un programa en la CPU puede ser más rápido, pero hay que limitar su ejecución.

Mecanismos para la ejecución directa:

Dual Mode Operativo - Modo de operación dual.
Privileged Instructions - Instrucciones Privilegiadas.
Memory Protection - Protección de Memoria
Timer Interrupts - Interrupciones por temporizador

Modo dual de operaciones

- Es un mecanismo que proveen todos los procesadores y algunos microprocesadores modernos.
- La arquitectura x86 provee 4 modos de operaciones vía el hardware, denominados rings.
- La **diferencia** de modos equivale a un **bit** en el **registro del procesador**. Esto permite diferenciar las instrucciones privilegiadas de las normales.

User & Kernel Mode

Kernel Mode: modo de operación 0, ejecuta instrucciones privilegiadas en nombre del kernel del SO

User Mode: modo de operación 3, ejecuta instrucciones en nombre del usuario, y en cada una el hardware chequea en qué modo de operación se encuentra

IOPL
Es un indicador que se encuentra en todas las CPU x86, ocupa los bits 12 y 13 en el registro FLAGS. Muestra los niveles de privilegio de E/S del programa o tarea actual.

Protección del Sistema

El modo dual permite que cada modo tenga su propio set de instrucciones, con lo cual el bit de modo de operación indica al procesador si una instrucción puede o no ser ejecutada. Esto se debe a que hay instrucciones que solo se pueden ejecutar en Kernel Mode, y no en User Mode.

Protección de Memoria

El sistema operativo y los programas que están siendo ejecutados por el mismo deben residir **ambos en memoria al mismo tiempo**. Para que un programa pueda ejecutarse debe estar en memoria. Debido a esto y para que la memoria sea compartida de forma segura, el sistema operativo debe poder configurar el hardware de forma tal que cada proceso pueda leer y escribir su propia porción de memoria.
Para esto, existen varios mecanismos de hardware:

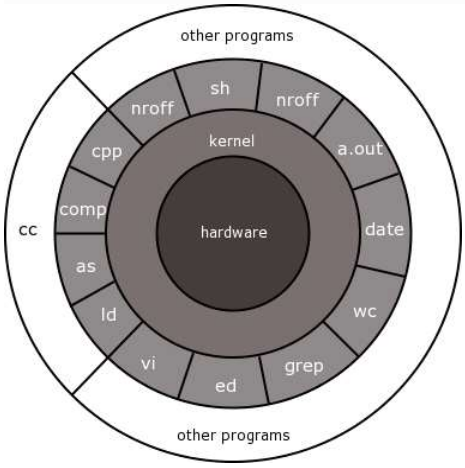
Timer Interrupts

Para que el kernel pueda tomar el control, debe haber algún mecanismo que le permita frenar un proceso en ejecución, y volver a tomar el control del procesador, y así de toda

la máquina.
Cada procesador posee un **hardware counter** integrado , si el procesador es multicore cada core tiene su propio contador de tiempo individual, el cual puede setearse para que luego de un determinado tiempo el procesador sea interrumpido. Cada CPU tiene su propio timer. Cuando esta interrupción ocurre, el proceso en modo usuario le transfiere el control al kernel.

Definición de Kernel

Si definimos al sistema operativo como el software que maneja y dispone de los recursos de una computadora, entonces el término kernel podría ser equivalente al sistema operativo. Debido a la existencia del Kernel, los programas son independientes del hardware subyacente ya que no interactúan directamente con él, sino que se comunican directamente con el kernel.



User Mode to Kernel Mode

Existen varias formas de alternar entre user mode a kernel mode

Interrupciones

Una interrupción es una **señal asincrónica** enviada al procesador para informarle de un **evento externo** que puede requerir su **atención**
El procesador chequea constantemente si una interrupción es lanzada. En caso de que ocurra, completa o frena la ejecución de cualquier instrucción que se esté ejecutando en ese momento, guarda el contexto en el que estaba esa instrucción, y arranca a manejar la interrupción en el kernel.

Excepciones del Procesador

Evento de hardware provocado por un programa de usuario. Funciona igual que las

interrupciones. Por ejemplo, intentar acceder fuera de la memoria del proceso, intentar ejecutar una instrucción privilegiada desde usermode, división por cero, etc.

Mediante la ejecución de Syscalls

Las system calls son funciones que permiten a los procesos de usuario pedirle al kernel que realice operaciones en su nombre. El kernel expone que estas funciones pueden ser utilizadas por un proceso a nivel usuario.

- Una system call **cambia** el modo del procesador de user mode a kernel mode, por ende la CPU podrá acceder al área protegida del kernel.
- El conjunto de system calls es fijo. Cada system call está identificada por un único número, que por supuesto **no es visible al programa**, éste sólo conoce su nombre.
- Cada system call debe tener un **conjunto de parámetros** que especifican información que debe ser transferida desde el **user space al kernel space**.

Kernel Mode to User Mode

Se puede dar de las siguientes formas:

Nuevo proceso

Cuando se inicia un **nuevo proceso** el Kernel copia el programa en la memoria, setea el contador de programa apuntando a la primera instrucción del proceso, setea el stack pointer a la base del stack de usuario y switchea a user mode.

Continuar luego de una interrupción, excepción o syscall

Una vez que el Kernel termina de manejar el pedido, este continúa con la ejecución de proceso interrumpido mediante la restauración de todos los registros y cambiando a user mode.

Cambio entre diferentes procesos

En algunos casos puede pasar que el Kernel decida ejecutar otro proceso que no sea el que se estaba ejecutando, en ese caso el Kernel carga el estado del proceso nuevo a través de la PCB y cambia a user mode.

Tipos de Kernel

Existen dos tipos de kernel

Kernel Monolítico

El kernel es un único programa (en realidad proceso) que se ejecuta continuamente en

la memoria de la computadora intercambiándose con la ejecución de los procesos de usuario.

Micro Kernel

El kernel sigue existiendo pero sólo implementa funcionalidad básica en el ring 0. Otros servicios se implementan en ring 1 o ring 2, estos servicios no son imprescindibles que se ejecuten en modo kernel exclusivamente.

Iniciar el sistema operativo y el Kernel.

El proceso de inicio de una computadora se divide esencialmente en 3 partes:

- Booteo.
- Carga del Kernel.
- Inicio de las aplicaciones de usuario.

Booteo

Este proceso es denominado bootstrap, y generalmente depende del hardware de la computadora. En él se realizan los chequeos de hardware y se carga el bootloader, que es el programa encargado de cargar el Kernel del SO.

Este proceso consta de 4 partes:

- Cargar el BIOS (Basic Input/Output System).
- Crear la Interrupt Vector Table y cargar las rutinas de manejo de interrupciones en modo real.
- La BIOS genera una interrupción 19 (INT 19) de la tabla de interrupciones.
- El paso 3 hace ejecutar el servicio de interrupciones, el handler de dicha instrucción, que es leer el primer sector de 512 bytes del disco a memoria y ahí termina.

Carga del Kernel

El BootLoader es un programa que se encarga de:

- Pasar a Modo Supervisor, esto es posible porque se realiza por hardware.
- Ir a buscar al kernel en el dispositivo donde está almacenado.
- Lo carga en la memoria principal.
- Setea el registro de PI (próxima instrucción).
- Ejecuta la primera instrucción del kernel.

Inicio de las aplicaciones de usuario

Una vez que el Kernel se ejecutó, las últimas operaciones que realiza son:

- Carga en memoria la aplicación que se debe ejecutar, normalmente es el shell.
- Setear el PI en la primera instrucción de esta aplicación.
- Pasar a modo usuario y dejar el control a la aplicación.

El Kernel es cargado normalmente como un archivo imagen, comprimido dentro de otro, esto quiere decir que no es un archivo ejecutable normal.
Contiene una cabecera de programa que hace una cantidad mínima de instalación de hardware, descomprime la imagen completamente en la memoria alta.
A continuación lleva a cabo su ejecución, para esto llama a la función startup del Kernel.

Fase de inicio del Kernel.

La función de arranque para el kernel, establece la gestión de memoria (tablas de paginación y paginación de memoria), detecta el tipo de CPU y cualquier funcionalidad adicional como capacidades de punto flotante, y después cambia a las funcionalidades del Kernel para arquitecturas no específicas de Linux, a través de una llamada a la función start_kernel().

Esta función (start_kernel) ejecuta:

- Establece el manejo de interrupciones (IRQ).
- Configura memoria adicional.
- Comienza el proceso de inicialización

Por lo tanto, el núcleo inicializa los dispositivos, monta el sistema de archivos raíz especificado por el gestor de arranque como de solo lectura, y se ejecuta Init que es designado como el primer proceso ejecutado por el sistema (PID=1).

En este punto, con las interrupciones habilitadas, el programador puede tomar el control de la gestión general del sistema, para proporcionar multitarea preventiva, e iniciar el proceso para continuar con la carga del entorno de usuario en el espacio de usuario.

El proceso de inicio

El trabajo de Init es “conseguir que todo funcione como debe ser” una vez que el Kernel está totalmente en funcionamiento. En esencia, establece y opera todo el espacio de usuario. Esto incluye:

- Comprobación y montaje de sistema de archivos.
- La puesta en marcha de los servicios de usuario necesarios y, en última instancia, cambiar al entorno de usuario cuando el inicio del sistema se ha completado.

En un sistema Linux estándar, Init se ejecuta como un parámetro (con valor entre 1 y

6) , conocido como nivel de ejecución y que determina qué subsistemas pueden ser operacionales. Cada nivel de ejecución tiene sus propios scripts que codifican los diferentes procesos involucrados, y son estas secuencias de comandos los necesarios en el proceso de arranque.

Después de que se han dado lugar todos los procesos especificados, Init se aletarga y espera a que sucede alguno de estos 3 eventos:

- Procesos comenzados finalicen o mueran.
- Un fallo de la señal de potencia (energía).
- Petición para cambiar el nivel de ejecución.

Procesos

Un programa es un conjunto de instrucciones y datos que esperan en algún lugar del disco para poder arrancar. Este conjunto de instrucciones y datos es tomado por el SO, quien mediante el Kernel lo transforma en algo útil.

El Sistema Operativo más precisamente el Kernel se encarga de:

- Cargar instrucciones y Datos de un programa ejecutable en memoria.
- Crear el Stack y el Heap
- Transferir el Control al programa
- Proteger al SO y al Programa

Un proceso es un programa en ejecución, el cuál incluye:

- archivos necesarios abiertos
- señales pendientes
- datos internos del kernel
- estado completo del procesador

- un espacio de direcciones de memoria
- uno o más threads, los cuales contienen:
    - un único contador de programa
    - un stack
    - un conjunto de registros
- una sección de datos globales

En el uso diario, se están corriendo muchos procesos al mismo tiempo y no debemos preocuparnos por si hay una cpu disponible o no, Esto se logra mediante la “ilusión” de que tenemos tantos cpus como queramos.
Esta ilusión se logra virtualizando la cpu. Corre un proceso, para arrancar otro y así se da la sensación de que hay muchos corriendo al mismo tiempo. Esto es llamado Time Sharing.

Virtualización de Memoria

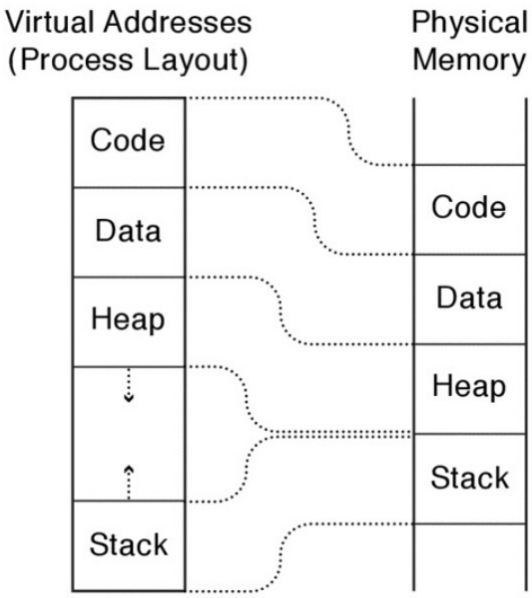
La virtualización de memoria le crea una ilusión al proceso haciéndole creer que tiene toda la memoria disponible para ser reservada y usada, como si fuera el único proceso ejecutándose en la computadora.

Está dividida en 4 segmentos:

- text instrucciones del programa
- data variables globales
- heap memoria dinámica alocable
- stack variables locales y trace de llamadas

Protección de Memoria

Para que un proceso pueda ejecutarse debe estar en memoria, mientras que el SO debe estar ahí para iniciar la ejecución, manejar las interrupciones y/o atender las syscalls.
El hardware debe proveer un mecanismo de protección de memoria. Uno de estos mecanismos es la **memoria virtual,** una abstracción por la cual la **memoria física** puede ser **compartida** por varios procesos.



Traducción de Direcciones

Se traduce una dirección virtual emitida por la CPU, en una dirección física (la memoria). Esto se realiza por hardware, por MMU - memory management unit.

Virtualización de Procesador

Consiste en dar la ilusión de la existencia de un único procesador para cualquier programa que requiera de su uso.

Simplicidad en la programación

- Cada proceso cree que tiene toda la CPU.
- Cada proceso cree que todos los dispositivos le pertenecen.
- Distintos dispositivos parecen tener el mismo nivel de interfaces.
- Las interfaces con los dispositivos son más potentes que el bare metal.

Aislamiento frente a Fallas:

- Los procesos no pueden directamente afectar a otros procesos.
- Los errores no colapsan toda la máquina.

Contexto de un Proceso

El contexto de un proceso consiste de:

- **User Address space:** dividido en text, data, stack, heap
- **Control Information:** kernel, 2 áreas para tener la info de control de un proceso, la **u área** y estructura **proc**. Cada proceso tiene su propio kernel stack y mapas de traducción de dirección
- **Credenciales**: incluyen los grupos IDs y user id asociados con el proceso
- **Variables de entorno:** conjunto strings heredadas del proceso padre
- **Hardware context:** contenido de registros de propósito general y un conjunto especial de registros del system (PC, SP, PWD, Memory management registers, registros unidad punto flotante)

Creación de un Proceso

La única forma de que un usuario cree un proceso en el sistema operativo UNIX es llamando a la system call fork.
El proceso que invoca a fork es llamado proceso padre, el nuevo proceso creado es llamado hijo.

¿Que hace fork?:

- Crea y asigna una nueva entrada en la Process Table para el nuevo proceso.
- Asigna un número de ID único al proceso hijo.
- Crea una copia lógica del contexto del proceso padre, algunas de esas partes pueden ser compartidas como la sección text
- Realiza ciertas operaciones de I/O.
- Devuelve el número de ID del hijo al proceso padre , y un 0 al proceso hijo

¿Qué hace fork(), el algoritmo:
La implementación de esta system call no es para nada trivial ya que cuando el proceso hijo inicia inicia a ejecutarse parece hacerlo casi en el aire:

- chequear que haya recursos en el kernel;
- obtener una entrada libre de la Process Table, como un PID único;
- chequear que el usuario no esté ejecutando demasiados procesos;
- macar al proceso hijo en estado “siendo creado”;
- copiar los datos de la entrada en la Process Table del padre a la del hijo;

- incrementar el contador del current directory inode;
- incrementar el contador de archivos abiertos en la File Table;
- hacer una copia del contexto del padre en memoria;
- crear un contexto a nivel sistema falso para el hijo;

Estados en Linux

Dado que en Linux los procesos son denominados tasks los estados de una task pueden ser:

- TASK_RUNNING (0): El proceso está o ejecutándose o peleando por CPU en la cola de run del planificador.
- TASK_INTERRUPTIBLE (1): El proceso se encuentra en un estado de espera interrumpible; este queda en este estado hasta que la condición de espera eventualmente sea verdadera, por ejemplo un dispositivo de I/O está listo para ser utilizado, comienza de su time slice, etc. Mientras el proceso está en este estado, cualquier señal (signal) generada para el proceso es entregada al mismo, causando que este se despierte antes que la condición de espera se cumpla.
- TASK_KILLABLE: este estado es similar al TASK_INTERRUPTIBLE, con la excepción que las interrupciones pueden ocurrir en fatal signals.
- TASK_UNINTERRUTPIBLE (2): El proceso está en un estado de ininterrupción similar al anterior pero no podrá ser despertado por las señales que le lleguen. Este estado es raramente utilizado.
- TASK_ STOPPED (4): El proceso recibió una señal de STOP. Volverá a running cuando reciba la señal para continuar (SIGCONT).
- TASK_TRACED (8): Un proceso se dice que está en estado de trace, cuando está siendo revisado probablemente por un debugger.
- EXIT_ZOMBIE (32): El proceso está terminado, pero sus recursos aún no han sido solicitados.
- EXIT_DEAD (16): El proceso Hijo ha terminado y todos los recursos que este mantenía para sí se han liberado, el padre posteriormente obtiene el estado de salida del hijo usando wait.

Scheduling

Debe existir un mecanismo que permita determinar cuánto tiempo de CPU le toca a cada proceso. Ese periodo de tiempo que el kernel le otorga a un proceso se denomina **time slice** o **time quantum**.

Time Sharing

Se refiere a compartir de forma concurrente un recurso de la computadora entre muchos usuarios.

Multi-programming

La idea del multiprogramming es **optimizar** lo *máximo posible el uso de la CPU*. Para esto, varios procesos pueden ejecutarse de forma concurrente en un único CPU/procesador, accediendo a los mismos recursos. Como hay un **solo procesador**, no existe la ejecución simultánea, entonces bajo el concepto de multiprogramming, el SO ejecuta parte de un programa, luego parte de otro, y así sucesivamente intercalando entre los diferentes procesos. El **Scheduler**, que forma parte del Kernel del SO, es el encargado de coordinar la forma en la que estos procesos se ejecutan.

Utilización de la CPU & Multi-Programming

Pongamos de ejemplo que un proceso gasta una **fracción p**, bloqueando en operaciones de input/output, si tenemos **n procesos** esperando para hacer operaciones de input/output, la probabilidad de que los **n procesos** estén haciendo input/output es **p^n**

Luego, la probabilidad de que se esté ejecutando algún proceso es 1 - **p^n** : **utilización de CPU**

Con un sólo proceso que tarda un 80% del tiempo en operaciones de input-output tenemos que el tiempo de utilización del CPU es de 1 - 0.8, que es el 20%. Con 3

procesos, este índice de utilización aumenta a 48%. Con 10 procesos, llega al 89%, dejando en claro la importancia de implementar multiprogramming.

First In, First Out (FIFO)

El algoritmo más básico para implementar como **política de planificaciones** es el FIFO.
Cuando comenzamos a trabajar en un proceso, seguimos su ejecución hasta que este finaliza.

- Es simple
- Es fácil de implementar
- Funciona bien para las suposiciones iniciales

Si se asume que todos llegan al mismo tiempo pero su duración es distinta puede suceder el efecto **Convoy Effect**. Sucede que cuando una tarea con poca duración viene después de una que tarda mucho tiempo, el sistema va a parecer ineficiente.

Shortest Job First (SJF)

Para resolver el Convoy Effect, se modifica la política para que se ejecute el proceso de duración mínima, luego, se ejecuta el proceso de duración mínima, y así sucesivamente.

- No se puede saber cuánto va a durar cada proceso
- Es pésimo para la variación en el tiempo de respuesta (Al realizar las tareas más cortas lo más rápido posible, SJF necesariamente realiza las tareas más largas lo más lentamente posible).

Shortest Time-to-Completion (STCF)

El scheduler puede adelantarse y determinar qué proceso debe ser ejecutado.
Entonces, cuando los procesos B y C llegan, se puede parar la ejecución del proceso A, decidir que se ejecuten los procesos B y C, y luego retomar la ejecución del proceso A.

Tiempo de Respuesta

El tiempo de respuesta surge con la aparición del time-sharing ya que los usuarios esperan una veloz interacción con la terminal de una computadora.
Nace como métrica el **response time**:

timeResponse = timeFirstRun - timeArrival

Round Robin (RR)

La idea del algoritmo es bastante simple, se ejecuta un proceso por un período

determinado de tiempo (slice) y transcurrido el período se pasa a otro proceso, y así sucesivamente cambiando de proceso en la cola de ejecución. Con Round Robin, no hay posibilidad de que una tarea nunca se ejecute; eventualmente llegará al frente de la cola y obtendrá su cantidad de tiempo

Lo importante de RR es la elección de un buen time slice, se dice que el time slice tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más. Una consideración es la sobrecarga: si tenemos un cuanto de tiempo demasiado corto, el procesador pasará todo su tiempo cambiando y realizando muy poco trabajo útil. Si elegimos un cuanto de tiempo demasiado largo, las tareas tendrán que esperar mucho tiempo hasta que obtengan un turno

Desventajas:

- Cuando varias tareas comienzan aproximadamente al mismo tiempo y tienen la misma duración. Round Robin rotará a través de las tareas, haciendo un poco de cada una, y terminando todas aproximadamente al mismo tiempo. ¡Esta es casi la peor política de programación posible para esta carga de trabajo! → FIFO y SJF son mejores en estos casos (La división de tiempo agregó gastos generales sin ningún beneficio).
- Dependiendo de la cantidad de tiempo, Round Robin también puede ser bastante pobre cuando se ejecuta una combinación de tareas vinculadas a E/S y vinculadas a cómputo. Las tareas vinculadas a E/S a menudo necesitan períodos muy cortos en el procesador para calcular la siguiente operación de E/S que se va a ejecutar. Cualquier retraso que se programe en el procesador puede provocar ralentizaciones en todo el sistema.

Multi-Level Feedback Queue (MLFQ)

Este planificador ha sido refinado con el paso del tiempo hasta llegar a las implementaciones que se encuentran hoy en un sistema moderno.
MLQF intenta atacar principalmente 2 problemas:

- Intenta optimizar el turnaround time, que se realiza mediante la ejecución de la tarea más corta primero, desafortunadamente el sistema operativo nunca sabe a priori cuánto va a tardar en correr una tarea.
- MLFQ intenta que el planificador haga sentir al sistema con un tiempo de respuesta
  interactivo para los usuarios por ende minimizar el **response time**; desafortunadamente los algoritmos como round-robin reducen el **response time** pero tienen un terrible **turnaround** time.

MLQF: Las reglas básicas

MLFQ tiene un conjunto de distintas colas, cada una de estas colas tiene asignado un nivel de prioridad.
En un determinado tiempo, una tarea que está lista para ser corrida está en una única cola.
MLFQ usa las prioridades para decidir cual tarea debería correr en un determinado tiempo

t0: la tarea con mayor prioridad o la tarea en la cola más alta será elegida para ser corrida.

Dado el caso que existan más de una tarea con la misma prioridad entonces se utilizará el
algoritmo de ***Round Robin*** para planificar estas tareas entre ellas.

Las 2 reglas básicas de MLFQ:

- Regla 1: si la prioridad (A) es mayor que la prioridad de (B), (A) se ejecuta y (B) no.
- Regla 2: si la prioridad de (A) es igual a la prioridad de (B), (A) y (B) se ejecutan en Round-Robin.

La clave para la planificación MLFQ subyace entonces en cómo el planificador setea las prioridades. En vez de dar una prioridad fija a cada tarea, MLFQ varía la prioridad de la tarea basándose en su comportamiento observado.

Primer intento: ¿Cómo cambiar la prioridad?

Se debe decidir cómo MLFQ va a cambiar el nivel de prioridad a una tarea durante toda la
vida de la misma (por ende en qué cola esta va a residir). Para hacer esto hay que tener en
cuenta nuestra carga de trabajo (workload): una mezcla de tareas interactivas que tienen un
corto tiempo de ejecución y que pueden renunciar a la utilización de la CPU y algunas tareas de larga ejecución basadas en la CPU que necesitan tiempo de CPU , pero poco tiempo de respuesta. A continuación se muestra un primer intento de algoritmo de ajuste de
prioridades:

- Regla 3: Cuando una tarea entra en el sistema se pone con la más alta prioridad
- Regla 4a: Si una tarea usa un time slice mientras se está ejecutando su prioridad
  se reduce de una unidad (baja la cola una unidad menor)
- Regla 4b: Si una tarea renuncia al uso de la CPU antes de un time slice completo
  se queda en el mismo nivel de prioridad.

Problema con este approach de MLFQ

- **Starvation**: si hay demasiadas tareas interactivas en el sistema se van a combinar para consumir todo el tiempo del CPU y las tareas de larga duración nunca se van a ejecutar.
- Un usuario inteligente podría reescribir sus programas para obtener más tiempo de CPU.

Segundo Approach ¿Cómo mejorar la prioridad?

Para evitar lo de starvation, simplemente se mejora la prioridad de todas las tareas en el
sistema:

- Regla 5: Después de cierto periodo de tiempo S, se mueven las tareas a la cola con más prioridad.

Con eso se garantizan 2 cosas:

- Se evita que lo procesos no se van a starve: al ubicarse en la cola tope con las otras
  tareas de alta prioridad estos se van a ejecutar en round-robin y por ende en algún
  momento recibirá atención
- Si un proceso que consume CPU se transforma en interactivo el planificador lo tratar
  como tal una vez que haya recibido el boost de prioridad.

¿Pero cuánto tiene que durar **S**?
Si el valor es demasiado alto, los procesos que requieren mucha ejecución van a caer en
starvation; si se setea con valores muy pequeños las tareas interactivas no van a poder compartir adecuadamente la CPU.
Se debe solucionar otro problema: Cómo prevenir que ventajeen (gaming) al planificador.

Cheating

La solución es llevar una mejor contabilidad del tiempo de uso de la CPU en todos los niveles del MLFQ.

Se modifica la regla 4a y 4b:

- Regla 4: Una vez que una tarea usa su asignación de tiempo en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce. (por ejemplo baja un nivel en la cola de prioridad)
  Resumen

Se vio la técnica de planificación conocida como multi-level feedback queue (MLFQ). Se puede ver porque es llamado así, tiene un conjunto de colas de multiniveles y utiliza feed
back para determinar la prioridad de una tarea dada. La historia es su guía: Poner atención
en cómo las tareas se comportan a través del tiempo y tratarlas de acuerdo a ello. Las reglas que se utilizan son:

- Regla 1: si la prioridad (A) es mayor que la prioridad de (B), (A) se ejecuta y (B) no.
- Regla 2: si la prioridad de (A) es igual a la prioridad de (B), (A) y (B) se ejecutan en
  Round-Robin.
- Regla 3: Cuando una tarea entra en el sistema se pone con la más alta prioridad
- Regla 4: Una vez que una tarea usa su asignación de tiempo en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce (Por ejemplo baja un nivel en la cola de prioridad).
- Regla 5: Después de cierto periodo de tiempo S, se mueven las tareas a la cola con más prioridad.

Planificación: Proportional Share

Es otro tipo de algoritmo de planificación, se basa en un concepto muy simple: en vez de
optimizar el turnaround o el response time, va a intentar garantizar que cada tarea obtenga
cierto porcentaje de tiempo de CPU.
También es conocido como **planificación por lotería** la idea básica es sencilla: cada tanto
se realiza un sorteo para determinar qué proceso tiene que ejecutarse a continuación, por
ende los procesos que deban ejecutarse con más frecuencia tiene que tener más posibilidades de ganar la lotería.

El concepto de la lotería es muy básico, se usan los **boletos** que representan cuánto se comparte de un determinado recurso para un determinado proceso. El % de los boletos que
un proceso tiene es el porcentaje de cuanto va a compartir el recurso en cuestión.

Mecanismo de los boletos

- Ticket currency: Existen como en la realidad distintos tipos de moneda y las tareas
  pueden tener los tickets comprados con distintos valores de moneda; el sistema

automáticamente los transforma en un tipo global de moneda

- Transferencia de boletos: Este mecanismo permite que un proceso temporalmente
  transfiera sus boletos a otro proceso. Este mecanismo es útil cuando se está utilizando la arquitectura cliente/servidor.
- Inflación: Con la inflación un proceso puede aumentar o disminuir la cantidad de boletos que posee esto lo puede hacer de forma temporal. Este método puede ser
  utilizado en un ambiente en el cual los procesos confían entre ellos, para que no haya avaros.

Implementación

Es muy fácil de implementar. Todo lo que se necesita es un buen generador de números
aleatorios que determine cuál es el número de la lotería ganador, una estructura de datos
para mantener la información de los procesos del sistema y finalmente un número total de
tickets.

En definitiva para tomar una decisión de planificación, se debe sortear un boleto; cuando se
tiene el número ganador se recorre la lista de procesos en busca del proceso que tenga ese
número.

Planificación multiprocesador

Este tipo de planificación es que se tiene múltiples núcleos de CPU empaquetados en un
único chip.
Esto conlleva un conjunto de dificultades:

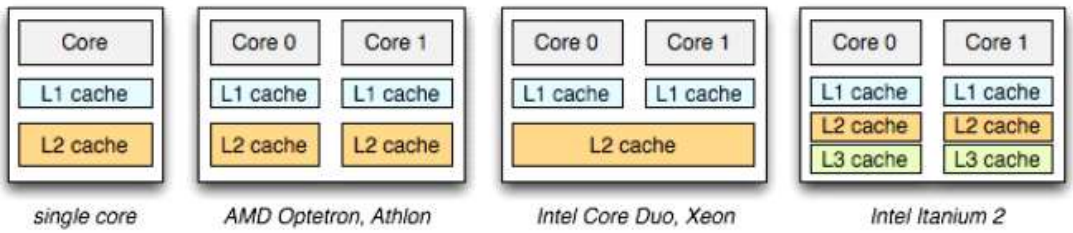
- Una aplicación típica usa únicamente una CPU:
  - ○ Entonces más CPU no va a implicar que corra más rápido.
  - ○ Las aplicaciones multi threads pueden diseminar el trabajo a lo largo de
    múltiples CPUs y por ende correr + rápido cuantas más CPU haya.
- La planificación en multiprocesadores es problemática.

Ley de Amdahl

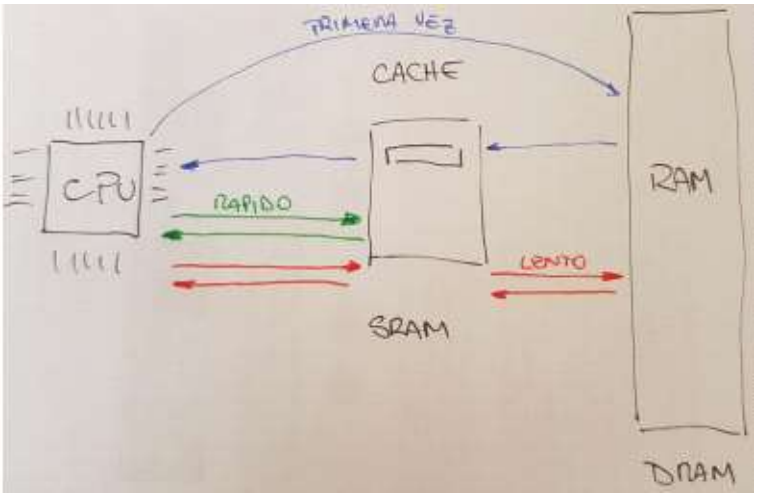
La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus
componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

La arquitectura multiprocesador

Hay que entender la diferencia entre hardware monoprocesador y multi-procesador, que se
centra básicamente alrededor de un tipo de hardware llamado caché,y de qué forma exactamente los datos en la caché son compartidos a través de los multiprocesadores. En un sistema con un único CPU hay una jerarquía de el hardware de caché que generalmente ayuda al procesador a correr los programas más rápidamente.



Cache y Localidad



La caché se basa en la noción de localidad, que hay 2 tipos:

- **Localidad temporal**: los programas tienden a acceder a direcciones de memoria **iguales** a las direcciones previamente accedidas.
- **Localidad espacial**: los programas tienden a acceder a direcciones de memoria **cercanas** a las direcciones previamente accedidas.

Temporal: se refiere a cuando la misma posición de memoria es referenciada muchas veces en un tiempo muy corto
Espacial: referencia de variables que están en posiciones contiguas de memoria

Coherencia de cache

¿Qué sucede cuando múltiples procesadores en un único sistema tiene que compartir una única memoria principal?

Ejemplo:

- Un programa en CPU 1 lee de memoria principal la dirección X, lo carga en caché y lo modifica (todavía no se actualiza en memoria principal)
- Se para la ejecución y se va a ejecutar otra en CPU2
- Como en la caché de CPU 2 no existe la dirección X, lo busca en la memoria principal → no está actualizado con el valor que se puso en CPU1.

Una solución (hay más pero son complicados):
Monitorización de los accesos de memoria, se puede hacer con un **bus**, usando una técnica llamada **Bus Snooping**:

- Cada caché pone atención en las actualizaciones de memoria mediante la observación del bus que está conectado a ellos y a la memoria principal. Cuando una CPU entonces ve que se actualizó un dato que está mantenido en su propio caché esta se va a dar cuenta de tal cambio y va a invalidar su copia o lo actualiza.

Afinidad de caché (cache affinity)

El concepto es básico:

- Cuando un proceso corre sobre una CPU en particular va construyendo un cachito
  del estado de si mismo en las cache de esa CPU
- Entonces la próxima vez que el proceso se ejecute sería bastante interesante que se
  ejecutará en la misma CPU, ya que se va a ejecutar más rápido si parte de su estado está en esa CPU.
- Si, en cambio, se ejecuta el proceso en una CPU diferente cada vez, el rendimiento
  del proceso va a ser peor, ya que tendrá que volver a cargar su estado o parte del
  mismo cada vez que se ejecute.

Por ende **un planificador multiprocesador debería considerar la afinidad de caché cuando toma sus decisiones de planificación**, tal vez prefiriendo mantener a un proceso
en un determinado CPU si es posible.

Planificación de cola única

La forma más fácil para tener un planificador para un sistema multiprocesador es la de reutilizar el marco de trabajo básico para un planificador de monoprocesador. Entonces se ponen todos los trabajos que tienen que ser planificados en una única cola, que se llamara SINGLE QUEUE MULTIPROCESSOR SCHEDULING o SQMS.
Esta forma de plantear la planificación tiene la ventaja de la simplicidad ya que no requiere mucho trabajo tomar la política existente que agarra la mejor tarea y la pone a ejecutar y adaptarla para que trabaje con más de una CPU. sin embargo, SQMS tiene sus limitaciones:

- No es **escalable**
- Los desarrolladores para asegurar que la planificación trabaje correctamente debe insertar algún tipo de **bloqueo en su código fuente**. Este bloqueo va a reducir en mucha la performance particularmente a medida que el número de la CPU del sistema empieza a crecer.
- ● La **afinidad de caché**. Teniendo en cuenta que cada CPU va a agarrar el próximo trabajo a ser ejecutado de la cola global compartida, cada tarea va a **terminar saltando de CPU en CPU**, haciendo exactamente lo opuesto de lo que recomienda la afinidad de caché. En algunos casos se podría proveer a ciertas tareas con cierta afinidad y a otras dejarlas cambiando de CPU para balancear la carga.

Planificación Multi-Queue

Debido a los problemas que tiene single queue scheduler varia gente optó por crear un planificador multi queue que se llama MULTI-QUEUE MULTIPROCESSOR SCHEDULING
en MQMS.

El esquema básico de planificación consiste en múltiples colas. Cada cola va a seguir una determinada disciplina de planificación, por ejemplo, round robin, cuando una tarea entra en el sistema ésta se coloca exactamente en una única cola de planificación, de acuerdo con alguna heurística. Esto implica que es esencialmente planificada en forma independiente.

El único problema de MQSM es el load imbalance. El load imbalance se da cuando una CPU queda ociosa frente a las demás que están sobrecargadas.

Para resolver esto la solución es mover las tareas de un lado a otro, esta técnica se conoce como migración o migration. Mediante la migración de una tarea a otra cpu se puede conseguir un verdadero balance de carga.

La vida real

Los Scheduler anteriores a Linux 2.6.0 eran O(n) a partir de ahí empezaron a tender a O(1).
El Scheduler 2.6 (Completely Fair Scheduler (CFS)) y sus revisiones introdujeron:

- el time-slice
- las colas de ejecución por proceso
- fair scheduling

Se implementa el fair-share scheduling de una forma altamente eficiente y escalable. Para lograr la meta de ser eficiente, CFS intenta gastar muy poco tiempo tomando decisiones de planificación, de dos formas :

- Por su diseño
- Debido al uso inteligente de estructuras de datos para esa tarea

Modo de operación básico

El objetivo de CFS es sencillo: dividir de forma justa la CPU entre todos los procesos que están compitiendo por ella. Esto lo hace mediante una simple técnica para contar llamada virtual runtime (Vruntime). El vruntime no es más que el runtime (es decir el tiempo que se está ejecutando el proceso normalizado por el número de procesos ejecutables.

A medida que un proceso se ejecuta este acumula vruntime. Cuando una decisión de planificación ocurre, CFS seleccionará el proceso con menos vruntime para que sea el próximo en ser ejecutado.

¿Cómo sabe el planificador cuando parar de ejecutar el proceso que está corriendo y correr otro proceso? **El punto clave aquí es que hay un punto de tensión entre performance y equitatividad:**

- si el CFS switchea de proceso en tiempos muy pequeños estará garantizando que todos lo proceso se ejecuten a costa de pérdida de performance, **demasiados**

context switches

- si CFS switchea pocas veces, la performance del scheduler es buena pero el costo
  está puesto del lado de la equitatividad (fairness).

La forma en que CFS maneja esta tensión es mediante varios parámetros de control:

- **sched_latency**: este valor determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar su switcheo (típico 48ms). CFS divide este valor por
  el número de procesos ejecutándose en la CPU para determinar el time-slice de un
  proceso, y entonces se asegura que por ese periodo de tiempo, CFS va a ser completamente justo.

Pero si hay muchos procesos ejecutándose esto no llevaría a muchos context switches y
por ende pequeños time slices? Sí, entonces surge:

- **min_granularity**: CFS nunca va a ceder el time-slice de un proceso por debajo de
  este número, por ende con esto se asegura que no haya overhead por context switch. Normalmente son 6 ms.

Weighting (Niceness)

CFS tiene control sobre las prioridades de los procesos, de forma tal que los usuario y administradores pueden asignar más CPU a un determinado proceso. Esto se hace con un
mecanismo llamado **nivel de proceso nice**, este valor va de -20 a +19, con un valor por
defecto de 0. Con una característica un poco extraña: los valores positivos de nice implican
una prioridad más baja, y los valores negativos de nice implican una prioridad más alta. CFS mapea el nice value con un peso, y estos permiten calcular efectivamente el time slice
para cada proceso.

Arboles Rojo-Negro

Uno de los focos de eficiencia del CFS está en la implementación de las políticas anteriores.
Pero también en una buena selección del tipo de dato cuando el planificador debe encontrar
el **próximo job a ser ejecutado**.

- Las **listas** no escalan bien **O(n)**.
- Los **árboles** sí, en este caso los árboles Rojo-Negro **O(logn).**

Algoritmo

Cuando el scheduler es invocado para correr un nuevo proceso, la forma en que el scheduler actúa es la siguiente:

- El nodo más a la izquierda del árbol de planificación es elegido (ya que tiene el tiempo de ejecución más bajo), y es enviado a ejecutarse.
- Si el proceso simplemente completa su ejecución, este es eliminado del sistema y del árbol de planificación.
- Si el proceso alcanza su máximo tiempo de ejecución o de otra forma se para la ejecución del mismo voluntariamente o vía una interrupción, este es reinsertado en el árbol de planificación basada en su nuevo tiempo de ejecución (vruntime).
- El nuevo nodo que se encuentre más a la izquierda del árbol será ahora el seleccionado, repitiendo así la iteración

Virtualización

Multiprogramming and Time Sharing

Esto nace de que la gente necesitaba compartir máquinas. Multiprogramming: múltiples procesos están listos para correr y el sistema operativo switchea entre ellos. Esto incrementa la utilización de CPU. Esto era una buena mejora cuando las máquinas eran muy costosas.

Aquí nace el time sharing, cuando la gente empieza a demandar más máquinas. Y en esto es donde la interactividad entre procesos se vuelve importante para el uso concurrente de una máquina por parte de varios usuarios.
Para que este proceso sea eficiente se comparte la memoria así solo se hace un Switch Context a altura de registros.

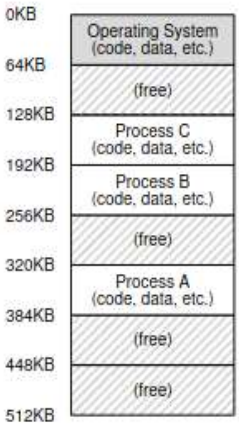


Figure 13.2: **Three Processes: Sharing Memory**

Aca se ven 3 procesos compartiendo memoria aunque solo uno está corriendo.
Los procesos van a correr según una política de scheduling.
A partir de esta propuesta toma valor la **protección** para evitar que los procesos interactúen entre sí de una forma no esperada

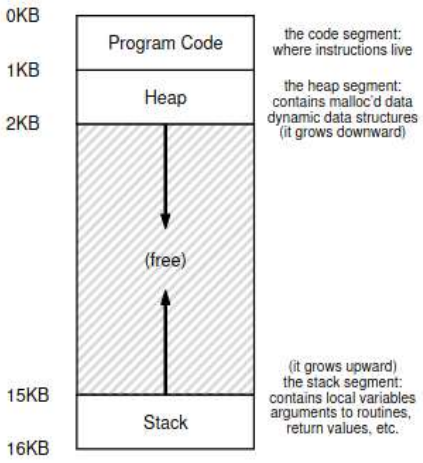


Figure 13.3: **An Example Address Space**

Address Space

Es la abstracción de la memoria física que facilita su uso por el OS y los procesos. Es la interfaz que ve el proceso de la memoria. Este address space contiene todo el estado de la memoria del proceso (code, stack, heap y otras cosas).

Esta es la disposición de un address space. Notar que stack y heap **crecen** en diferentes direcciones y el code vive en la parte alta de la memoria de un proceso.

De todas maneras esta forma de ver el proceso es una forma proveída por el sistema operativo como se menciona más arriba. Este **virtualiza** la memoria haciendo creer al programa que este está alojado desde la dirección física 0 y 16k. Estas direcciones son direcciones **virtuales**.

En el fondo el sistema operativo debe traducir direcciones de memoria virtuales a direcciones de memoria físicas para poder hacer el engaño. Esta virtualización hace lo más simple posible el trabajo de un usuario, pero complejiza bastante el trabajo del sistema operativo, quien la tiene que lograr de una manera eficiente en tiempo de traducción y espacio de aloje de mecanismos de traducción. También es la encargada de la protección de los procesos y del SO, ante la acción de otros procesos (load, store, fetch).

API de memoria

Tipos de memoria

- **Memoria de Stack:** Es la memoria en el stack de un proceso, se asigna con una simple asignación de variable
- **Memoria de Heap:** Es la memoria obtenida y liberada activamente por el programador, se utiliza malloc(size), esta función devuelve un puntero (void*) a la memoria alocada o null en caso de error. Usa free(puntero) para liberar.

Implementación de malloc(): se usa la syscall brk, que cambia el lugar donde el heap termina o se “rompe”(break). brk recibe la dirección de memoria nueva de fin de heap. También puede usar sbrk() que hace lo mismo que brk pero con una cantidad de memoria contigua.

Address Translation

Mediante la técnica Hardware Based Address Translation o como más comúnmente se conoce Address Translation.
Este proceso transforma una dirección virtual en una dirección física. El hardware por sí solo no lo hace, pero si provee un mecanismo de bajo nivel eficiente.
El sistema operativo se involucra en setear el hardware de forma correcta, gerenciar la memoria e intervenir con criterio sobre la misma.
Una vez pasa esto se realiza la ilusión de que cada programa tiene su memoria propia y privada, aunque en la realidad esto no es así, los programas pueden estar compartiendo memoria.

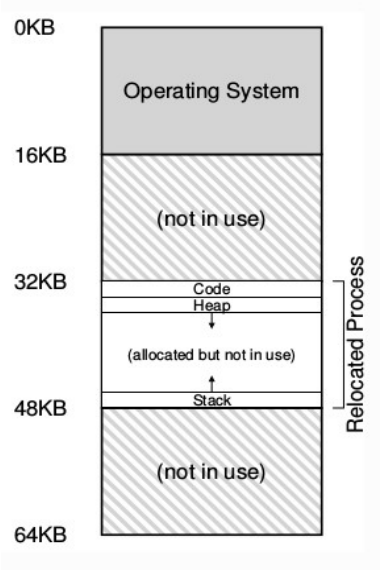
Este mecanismo es proporcionado por el hardware y vigilado por el sistema operativo. Lo que el SO debe vigilar es que se mantenga registro de la parte libre, en uso y como la memoria es utilizada.

Este proceso es completamente transparente aunque la memoria se almacena en otro lugar del que se pide. El translate es como una función.

Características de address translation

- **Process isolation:** evita que un proceso pueda tocar otro o el mismo kernel. Construye una sandbox para cada proceso.
- **Interprocess communication**: coordina procesos para que estos compartan memoria de manera eficiente.
- **Shared code segments:** permite que varios procesos compartan program instructions.
- **Program initialization**: mediante address translation se puede iniciar un programa luego de cargarlo.
- **Efficient dynamic memory alloc:** El mecanismo ayuda a alocar memoria en casos borde donde el heap o stack crecen mucho.

- **Cache management**: permite al SO que disponga de que va en la caché en función de su posición física para eficiencia.
- **Program debugging:** El mecanismo ayuda a prevenir un buggy program que sobreescriba su propio código e instalar breakpoints.
- **Efficient I/O:** ayuda a tener memoria para el I/O, que es mucho más rápido que el disco.
- **Memory mapped files:** permite mapear archivos en un address space de forma eficiente.
- **Virtual memory**: hace creer a un proceso de que dispone de más memoria de la que realmente dispone.
- **Checkpoint and restart:** Provee checkpoints por crashes para programas largos.
- **Persistent data structures**: El SO puede proveer una abstracción de una región de memoria persistente, ante un problema de crashes de programa.
- **Information flow control**: Es la base para controlar el flujo, permitiendo capturar filtrados de información de entrada y salida del sistema.
- **Distributed shared memory**: Se puede levantar una red de servers para compartir memoria a gran escala.



Esta es la **disposición real** de un proceso en memoria física (aunque él crea que está en la dirección 0).

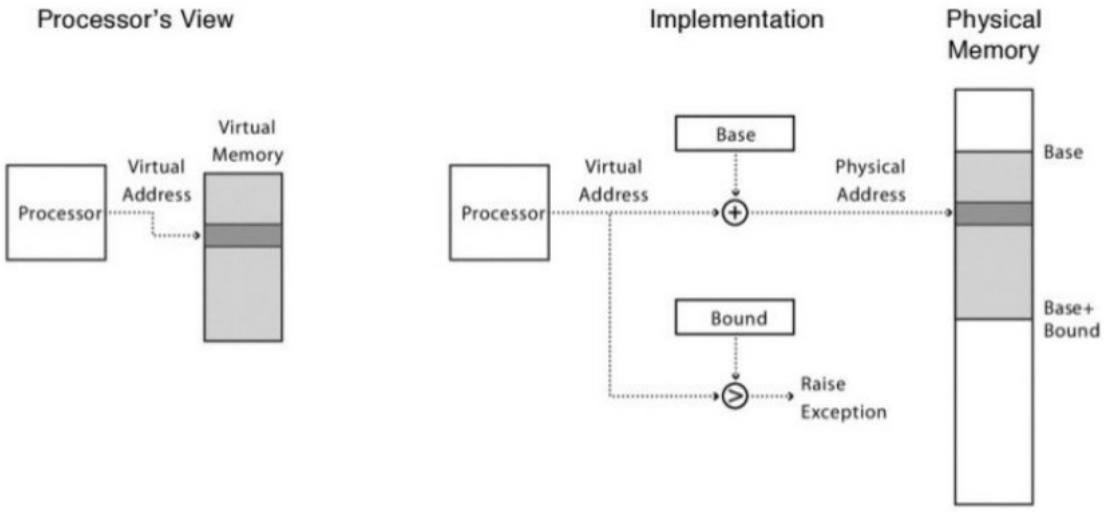
Dynamic Realocation o Memória Segmentada

Base and Bound

La primera implementación de address translation, es una idea llamada base y segmento. Estos dos componentes están en dos registros de la CPU y permiten que un address space pueda ser ubicado en cualquier lugar deseado de la memoria física, y el SO garantiza que el proceso solo puede acceder a su address space.

Los programas se escriben y compilan como si fuera cargado en la dirección física, cuando se ejecuta se carga la base en un registro. Luego cuando el proceso se ejecuta las referencias son de la siguiente manera:

physical address = virtual address + base

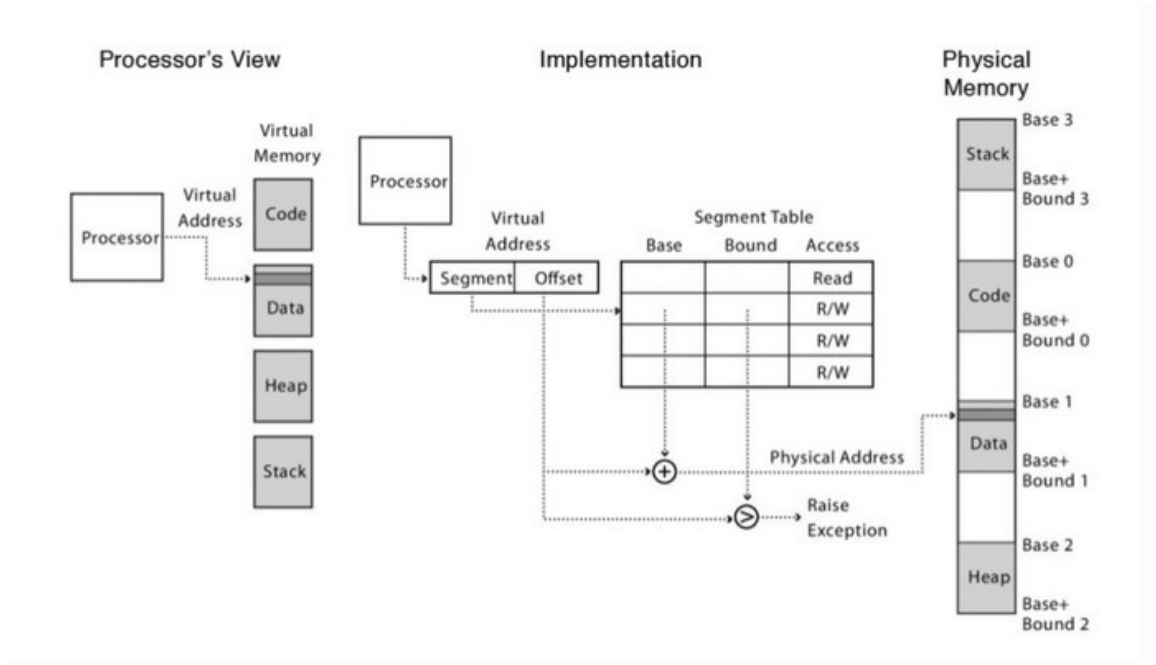


Debido a que el address space se podría mover inclusive durante tiempo de ejecución el método se llamó dynamic realocation.

Aparte de la base también se tiene un registro de bound que ayuda a la protección. El procesador chequea que estés dentro del rango entre base y bound para que sea legal. Si es mayor al bound, exception raise y fin de proceso.
Estas estructuras se mantienen dentro del procesador y se administra por la MMU.
El bound puede almacenar el largo del segmento o la posición final del mismo, son dos implementaciones válidas.
Internal Fragmentation es cuando un segmento tiene espacio libre dentro de sí.

Address Translation con Tabla de Segmentos

La técnica anterior solo deja tener un par de base bound, así que se implementa un arreglo con todos los pares que necesite. Cada entrada del arreglo controla una porción del virtual address space. La memoria física de cada segmento se almacena continuamente, pero los segmentos pueden estar en distintas partes de la memoria física.



Las direcciones virtuales se componen de número de segmento y un offset del mismo.

Bound se chequea contra la suma de **base + offset** y así chequea el límite.

Segmentation fault era el error que tiraba en un una máquina con segmentación cuando se quería trabajar con memoria fuera de límites.

Detalles:

- Se necesita saber si se está trabajando con un stack por su diferente forma de crecimiento
- Si se permite leer, ejecutar y/o escribir el segmento
- Si la segmentación es de grano fino o grueso (muchos segs chiquitos o pocos segs grandes).

El problema de la segmentación es la fragmentación, precisamente la fragmentación externa que significa que la memoria se llena de agujeros libres y complica el alocado de memoria contigua y el crecimiento de la memoria de un proceso.
La solución a ese problema es compactar la memoria física reorganizando los segmentos, si un proceso está corriendo llevar esos datos a una memoria más grande que sea contigua. Esto es algo muy pesado computacionalmente.

Memoria Paginada

Introducción

Page Frame: pedazos fijos de memoria física, de 4k.

La primera **diferencia** entre memoria paginada y memoria segmentada es que hay una page table para cada proceso que sus entradas contienen punteros a diferentes page frames o a otras pages tables si es una traducción multinivel.

Los pages frames están alineados a potencias de 2 para poder acceder más **fácil y eficientemente**. Otra **diferencia** es que gracias a esto ya no se necesita el bound de la segmentación.

La paginación **resuelve la principal limitación de la segmentación**: alocar memoria libre es muy fácil ya que el sistema operativo representa la memoria como un bitmap y alcanza con chequear si un bit está ocupado o no. Además, la memoria virtual es lineal.

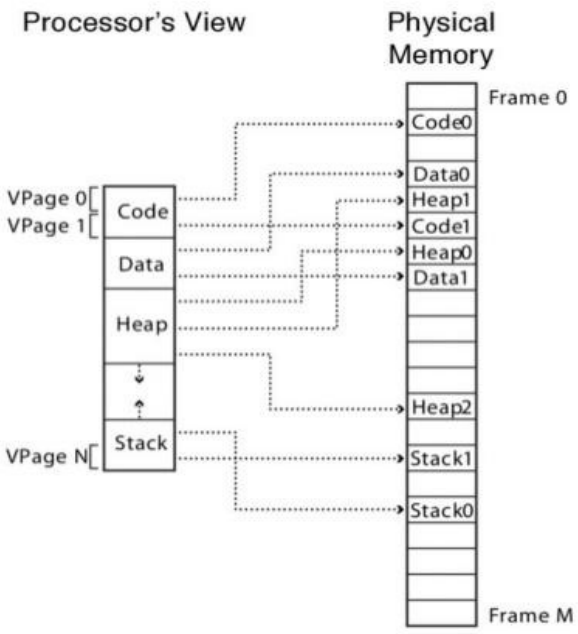
core map (primera vez que lo escucho, quizá no entra): guarda información acerca de cada frame físico el cual es apuntado por page tables. Esto es útil cuando se tiene más de un proceso, ya que más de un proceso podría estar accediendo al mismo pageframe desde diferentes page tables entries.

Una feature que **permite la page table y no la segmentación** es que un programa puede empezar a correr antes de que todo el código y la data fuera cargado en memoria. El OS marca todas las páginas como inválidas y las va habilitando a medida que las páginas son recuperadas, pero en el mientras tanto, se puede iniciar la ejecución con la particularidad de que el hardware causará una excepción si se intenta acceder a una región que no ha sido cargada.

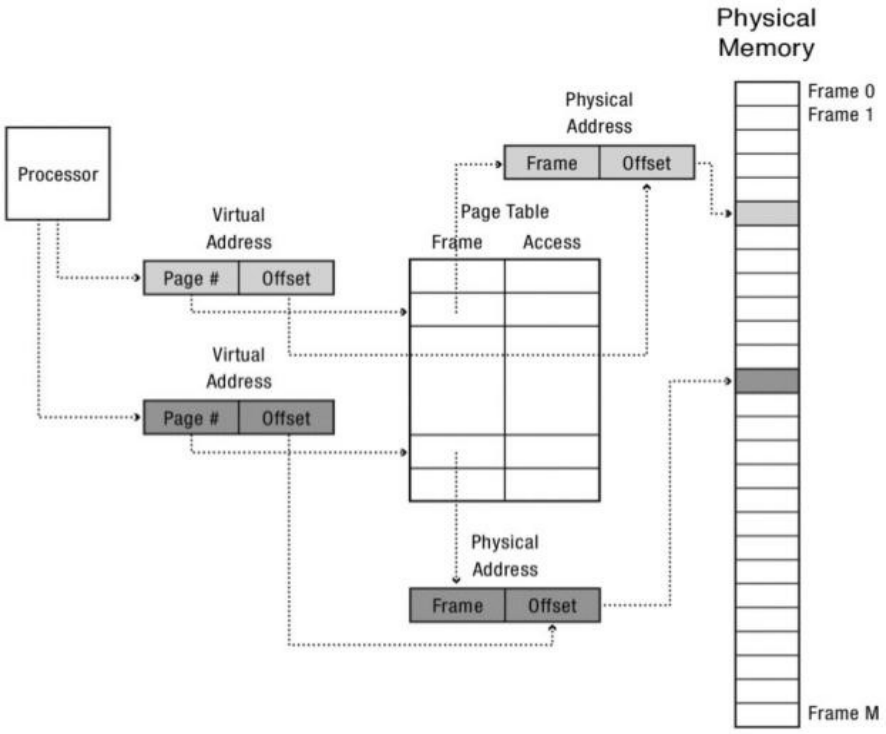
Protección: otra **ventaja** es que las page tables pueden tener bits de protección para poder decir cuando un programa puede o no, acceder a la página física que está buscando. Por ejemplo puede ser de modo lectura y nadie la podría modificar, salvo el kernel.

Una **desventaja** es que si bien la gestión de la memoria física se vuelve sencilla, la gestión de la memoria virtual se vuelve muy compleja, ya que para compiladores o call procedures, esperan que la memoria virtual sea contigua.

Otra **desventaja** es que la page table es proporcional al tamaño de la memoria virtual, por lo cual a grandes memorias, quedan matrices muy dispersas que generan sobrecarga de trabajo a la hora de ser utilizadas. Podría **reducirse el espacio** de la page table usando páginas más grandes, pero esto sería un problema de todas formas porque generaría fragmentación interna. **Una solución** a este problema es el uso de **paginación multinivel**, donde cada una de las page table entry del primer nivel de paginación, **apunta a otras páginas** del mismo tamaño y así sucesivamente hasta que se considere que la relación páginas/tamaño memoria virtual es correcta.



Address Translation con Page Table



En este caso donde la paginación es de un nivel, la memoria virtual se divide de la siguiente forma:

- El número de la página virtual es el índice de la page table.
- La page table contiene un número de pageframe y permisos.
- Si quien intenta acceder tiene los permisos necesarios, obtiene el número de pageframe y lo concatena con el offset de la página virtual.
- Con este número final, accede a la dirección física que necesita.

Multi-Level Address Translation

Ventajas vs Paginación

Para esto dejan de usarse bitmaps y pasan a usarse árboles o hashes, o ambos, para una traducción más eficiente. Porque permiten acceder a memoria física en O(log(n)), recorrer el array de entradas sería muy costoso en tiempo de procesamiento.

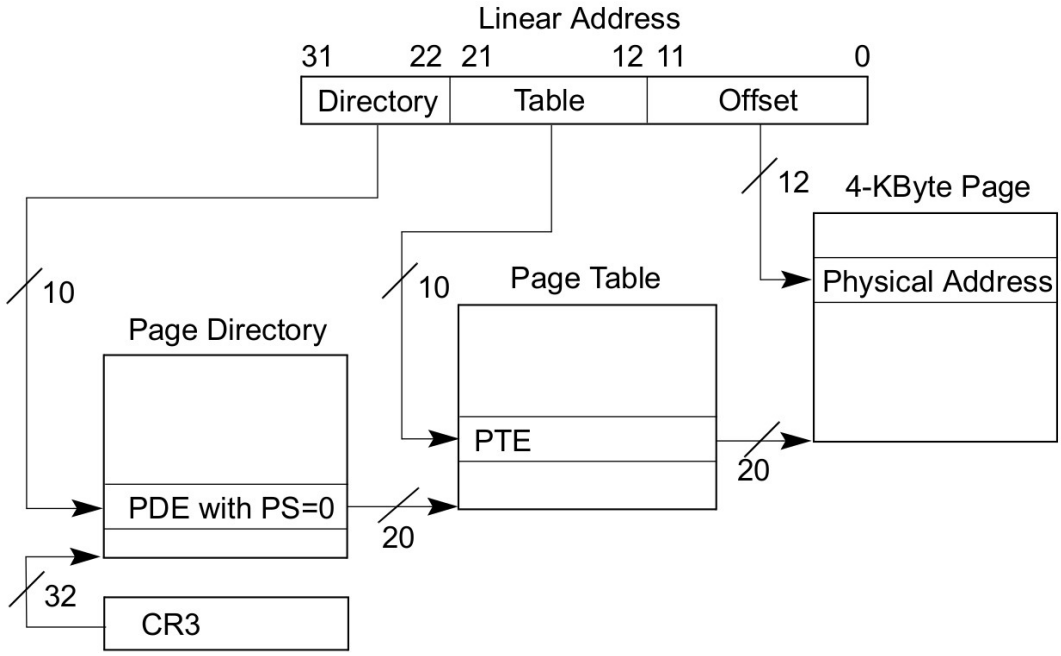
Una **ventaja** de la traducción de memoria (**búsqueda eficiente, efficient lookup**), es

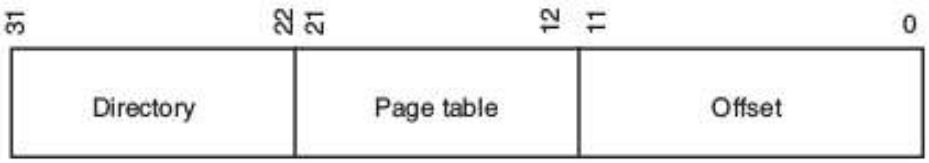
que la memoria virtual es lineal pero la física puede no serlo. Que sea lineal permite que direcciones consecutivas van a poder ser recuperadas con facilidad por la traducción, y estas direcciones recuperadas pueden estar muy dispersas en la memoria física.

Otras **ventajas:**

- **alocado eficiente de la memoria**: lo mencioné arriba, falta aclarar en donde…
- **transferencia eficiente de disco:** se simplifica la escritura en disco dado que las páginas son múltiplos de los sectores de disco, los cuales son usados para particionarlos.
- **búsqueda eficiente reversa:** consistente en recuperar una dirección virtual dada una página física.
- **protección de páginas**: lo mencioné arriba como una ventaja.

Paginación con Page Directory:



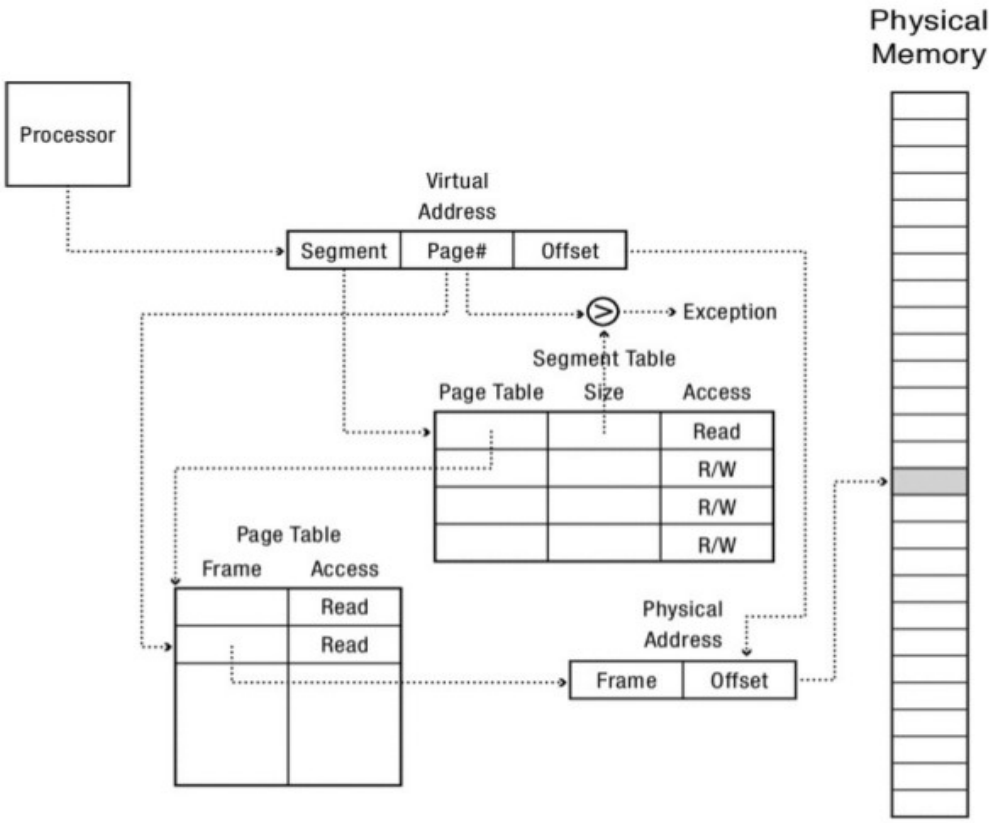


(a)

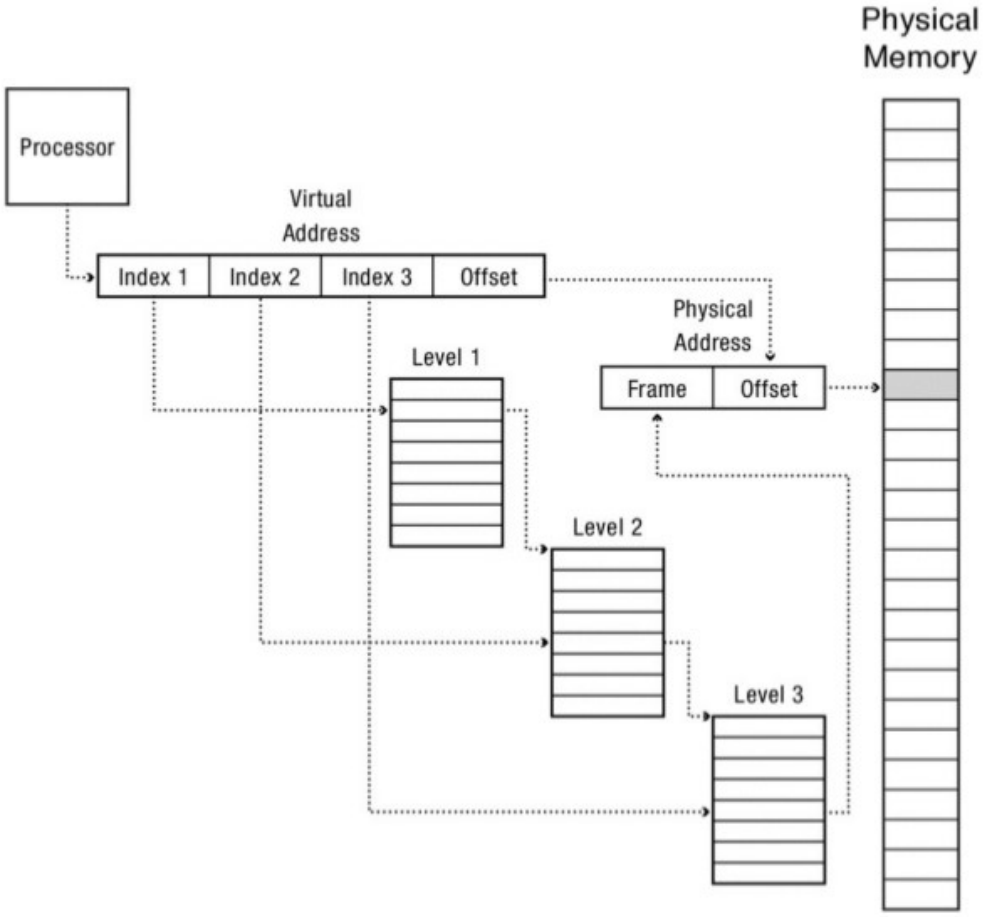
Donde **los primeros 12 bits** son del offset, que es lo que se usará para posicionarse dentro de un pageframe. **Los siguientes 10 bits** se usarán para ubicar el número del index de la page table.

PS: page small

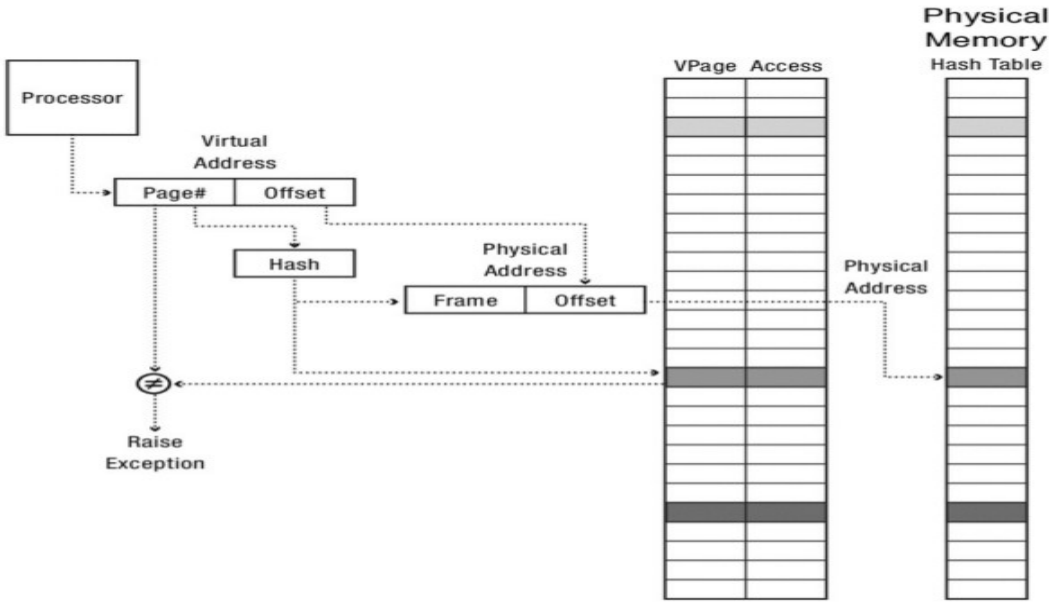
Segmentación Paginada



Address Translation con 3 niveles de Page Tables



Address Translation con Tabla de Hash por Software



Hacia una eficiente Address Translation

El problema de los múltiples niveles de paginación, es que en cierto punto se pierde mucho tiempo yendo a buscar una dirección física cuando los niveles tienen cierta profundidad. La solución a este problema es usar **caché**.

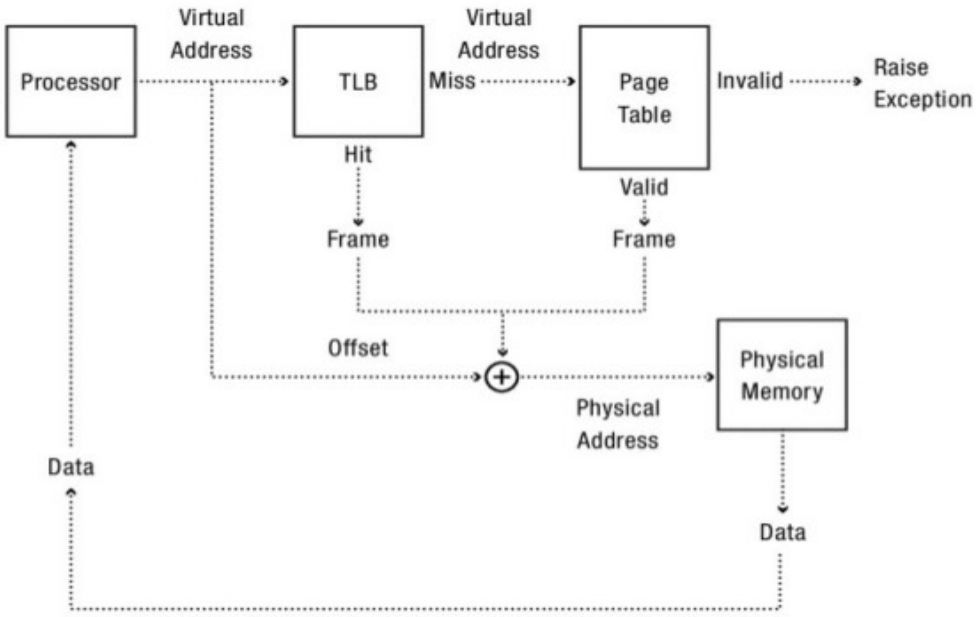
El mecanismo utilizado para cachear esto es **TLB Translation-Lookaside Buffer**. Es utilizado para evitar los tiempos de traducción de tantas direcciones virtuales, **forma parte de la MMU** y solamente cachea direcciones físicas, como resultado de una dirección virtual.

Es una tabla a nivel hardware, de la siguiente forma:

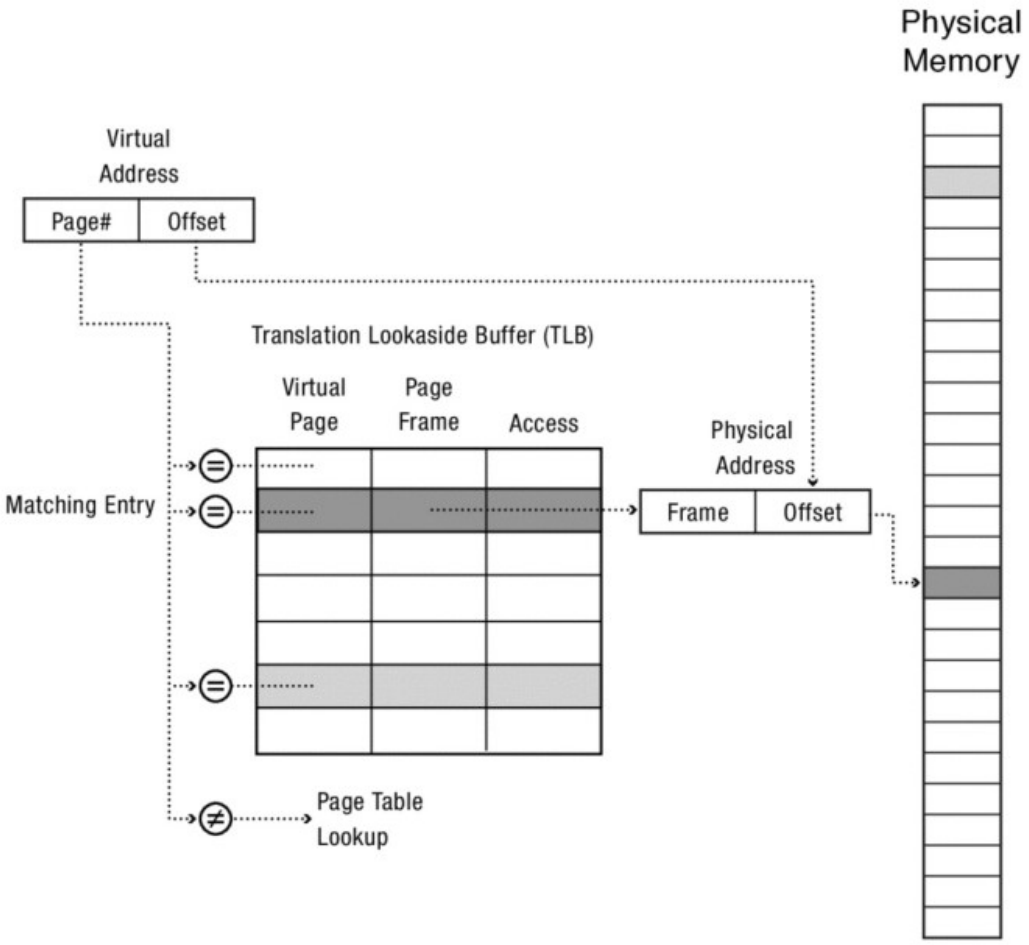
```
TLB entry = {
  virtual page number,
  physical page frame number,
  access permitions
}
```

TLB HIT: existe una dirección virtual que matchea con la que se intenta traducir.

TLB MISS: no existe una dirección virtual que matchea con la que se intenta traducir.



Las traducciones deben ser rápidas, por lo cuál la memoria debe ser chica. De otra manera se estaría replicando lo que hace la traducción de memoria convencional.



Localidad Espacial

Cuando hay un miss se accede a la página física que se trajo en el momento de guardar la dirección virtual en la caché. Esto lo que genera por principio de localidad espacial, es que los elementos cercanos al primer miss (y almacenamiento en la TLB) sean accedidos de forma más rápida. Como en el siguiente ejemplo:

De esta forma hay solamente 1 miss por cada salto de fila… Entonces se ahorran muchos accesos a memoria.

```
for (int i = 0; i <= 1000; i++) {
    for (int j = 0; j <= 1000; j++) {
        printf(arr[i][j]);
    }
}
```

En cambio el mismo for, con un:
print(arr[j][i]); genera que haya un miss por cada acceso, ya que las posiciones están claramente muy alejadas unas de otras.

Consistencia de la TLB

El sistema operativo garantiza que cada programa solo vea su memoria.

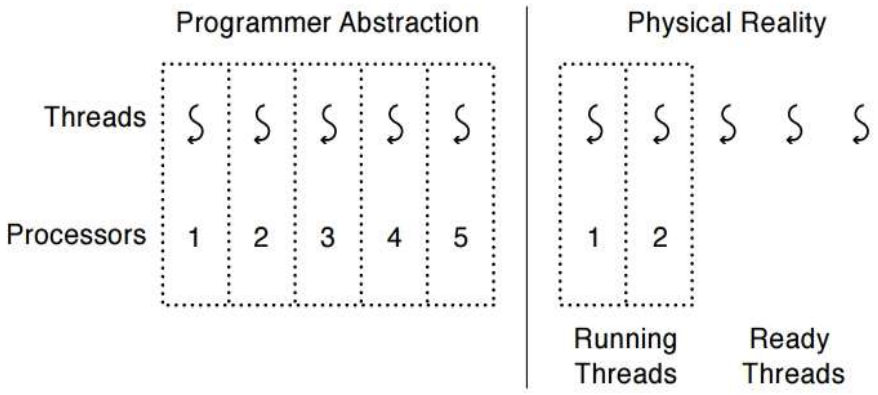
Context Switch

Se descartó la TLB, se hace un **flush de la TLB**. Esto es que la TLB pasa a ser invalidada para que el proceso entrante no pueda ver las direcciones físicas actuales.

Cuando el sistema operativo modifica una entrada en una page table, el mismo sistema operativo debe encargarse de dejar la TLB en un estado seguro. En un sistema multiprocesador cada procesador puede tener una copia de una transacción en su TLB, en este caso, la correspondiente entrada de cada una de las TLB son descartadas. Esto lo hace el propio procesador desencadenada por una señal del SO para que la borren.

Concurrencia

La concurrencia se refiere a un **conjunto de actividades** que pueden suceder **al mismo tiempo** e **interactuar con los mismos recursos.**
El concepto clave es escribir un programa concurrente como una secuencia de streams de ejecución o threads que interactúan y comparten datos en una manera precisa



La abstracción:

Un **thread** es una secuencia de ejecución atómica que representa una tarea planificable de ejecución. Los threads no se enteran de las interrupciones, creen que siempre se están ejecutando.

Secuencia de ejecución atómica: cada thread ejecuta una secuencia de instrucciones tal como lo hace un bloque de código en programación secuencial
Tarea planificable de ejecución: El SO tiene injerencia sobre el mismo en cualquier momento, puede ejecutarlo, suspenderlo y continuarlo cuando desee.

Thread VS Procesos:

Thread:

- ID
- Valores de registros
- Stack propio
- Política y prioridad de ejecución
- Errno propio
- Datos específicos del thread (estado ,etc)

Se pueden dar los siguientes casos:

- **One thread per process:** un proceso con una única secuencia de instrucciones ejecutándose de inicio a fin (clásica programación secuencial)
- **Many threads per process:** programa visto como threads ejecutándose dentro de un proceso con derechos restringidos. En un tiempo ti dado pueden estar algunos corriendo y otros suspendidos. Ejemplo en I/O kernel desaloja algunos threads para atender la interrupción y al terminar los vuelvo a correr.
- **Many single-threaded process:** limitación de algunos SO que permitan muchos procesos pero 1 thread por proceso
- **Many kernel threads:** el kernel puede ejecutar varios threads en kernel mode

Thread Scheduler:

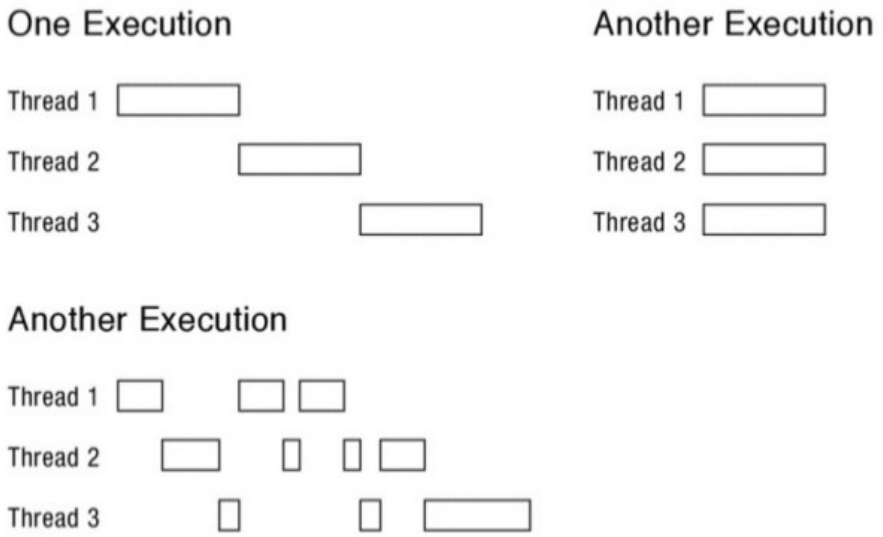
Necesario para dar la ilusión de que hay muchos threads corriendo con un número fijo de procesadores.
El cambio entre threads es transparente (el programador no se preocupa por ello)

Los threads proveen un modelo de ejecución en el que **cada thread corre en un procesador virtual dedicado con una velocidad variable e impredecible.**

Cada thread cree que las instrucciones se ejecutan siempre una detrás de otra sin interrupciones. Entonces se pueden dar múltiples escenarios de ejecución. Por ejemplo:

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Además pueden entrelazarse los distintos threads de distintas formas, como por ejemplo:



Los threads pueden relacionarse entre sí de dos formas:

- **Multi-threading Cooperativo:** no hay interrupción salvo a que se solicite
- **Multi- threading Preemptivo:** + usado, un thread en estado running puede ser movido en cualquier momento.

API de threads:

Creación de un thread:

```
#include<pthread.h>
int pthread_create(pthread_t * thread, const pthread_att_t * att,
        void * (start_routine) (void *), void * arg)
```

thread: puntero a la estructura para interactuar con el thread
att: especificar atributos que tiene el thread (suele ser NULL)
start_routine puntero a función void donde arranca el thread
arg: puntero a void de argumentos de la función
retorna: 0 en éxito, otro valor si hubo error

Terminación de un thread:
hay que esperar a que finalice su ejecución, usa la función:

```
int pthread_join(pthread_t thread, void **value_ptr )
```

thread es el thread que esperamos
value_ptr es un puntero al valor esperado de retorno (si no se devuelve nada NULL)

Estructura y ciclo de vida de un Thread

Para que la ilusión sea creíble hay que guardar y cargar el estado de cada thread. Además como cada thread puede correr en el procesador o en el kernel, también deben haber estados compartidos que no cambian entre modos.
Hay 2 tipos de estados:

- Estado **per thread**
- Estado **compartido** entre varios threads

Estado Per-thread y TCB

Cada thread debe poseer una estructura que representa su estado. Esta estructura es el **TCB (thread control block),** se crea una entrada x cada thread y almacena su estado per_thread:

- El estado de computo que debe ser realizado por el thread

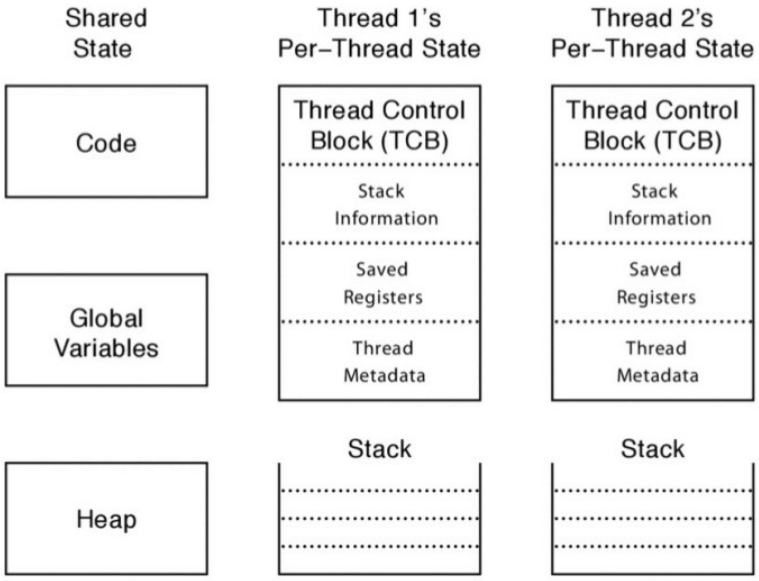
Para poder parar y arrancar muchos threads debe poder guardar el estado actual del bloque de ejecución, lo que incluye:

- Puntero al stack del thread
- copia de sus registros en el procesador

Metadata de un thread:

Para cada thread se debe guardar:

- ID
- Prioridad de scheduling
- Status



Estado compartido:

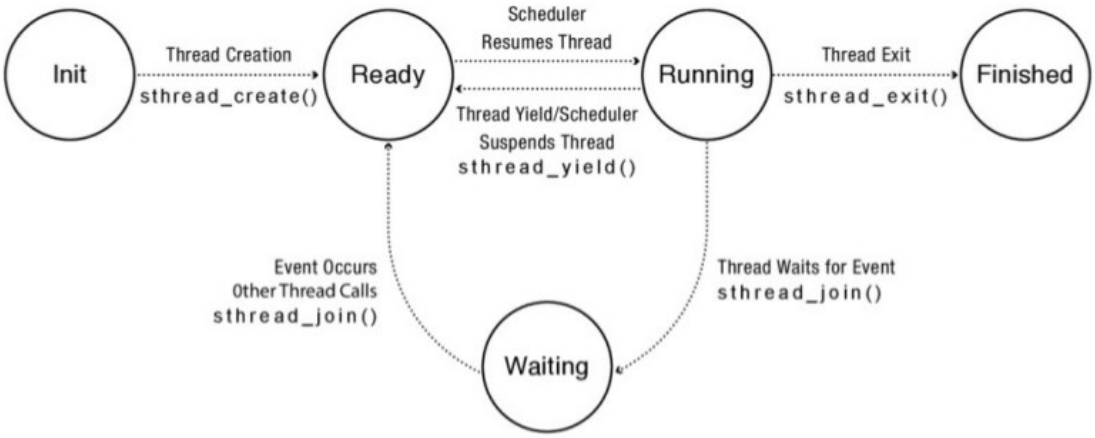
Aquí se guarda la información compartida por varios threads, esta es:

- El código
- Variables globales
- Variables del heap

Estados de un thread:

- **Init:** está en este estado mientras se está inicializando el estado per-thread y se está reservando la memoria para las estructuras. Cuando esto finaliza el estado será **ready**. También se lo pone en la lista **ready list** en donde todos los threads listos esperan a ser ejecutados
- **ready:** está listo para ser ejecutado pero todavía no lo está siendo en ese momento. TCB en la ready list y los registros ya están en la TCB. En cualquier momento el thread scheduler puede cambiarle su estado a **running**
- **running** Está siendo ejecutado en ese instante por el procesador. Valores de los registros en el procesador. De acá puede pasar a Ready de dos formas:
    - scheduler lo pasa mediante el desalojo o preemption guardando los valores de los registros y cambiando el thread q se esta ejecutando por

el próxima de la lista de ready
    - voluntariamente el thread puede solicitar abandonar la ejecución usando thread_yield por ejemplo
- **waiting:** thread está esperando a que suceda algún evento.Un thread en **waiting** no puede pasar a **running** directamente, entonces se guardan en la **waiting list.** Cuando el evento ocurre el scheduler cambia su estado de **waiting** a **ready**, moviendo la TCB de la waiting list a la ready list.
- **finished:** Un thread en estado finished nunca más volverá a ser ejecutado. En la lista finished lista están las TCB de los threads ya terminados.



Threads y Linux:

Diferencias Proceso/Thread

Los **threads**

- Por defecto comparten memoria
- Por defecto comparten los descriptores de archivos
- Por defecto comparten el contexto del filesystem
- Por defecto comparten el manejo de señales

Los **procesos**

- Por defecto no comparten memoria
- Por defecto no comparten los descriptores de archivos
- Por defecto no comparten el contexto del filesystem
- Por defecto no comparten el manejo de señales

Linux utiliza un **modelo 1-1** (proceso-thread), con lo cual **dentro** del **kernel no existe**

distinción alguna entre thread y proceso, todo es una tarea ejecutable.

Syscall clone()

Tanto fork como pthread_create utilizan a la syscall clone, pero de distinta forma.

Sincronización:

En el modelo multihilo de programación hay 2 escenarios posibles:

- Programa compuesto por múltiples threads independientes que trabajan sobre datos completamente separados entre sí e independientes
- programa compuesto por múltiples threads que trabajan en forma cooperativo sobre un set de memoria y datos compartidos

El primer caso, no requiere diferencias respecto a cómo se trabaja en la programación secuencial

La forma secuencial no sirve en un modelo de threads cooperativo ya que:

- ejecución del programa depende de cómo se intercalan los threads
- ejecución de un programa no es determinista
- compiladores y procesador físico pueden reordenar las instrucciones

Por todo esto la programación multithreading puede incorporar bugs que se caracterizan por ser:

- sutiles
- no determinísticos
- no reproducibles

Con el fin de evitarlo se debe estructurar el programa para facilitar el razonamiento concurrente y utilizar primitivas estándares para sincronizar el acceso a recursos compartidos.

Race Conditions:

Una Race condition se da cuando el resultado de un programa depende en como se intercalaron las operaciones de los threads que se ejecutan en el proceso.

Operaciones Atómicas:

Una operación atómica es una operación que no puede dividirse en otras y se garantiza la ejecución de la misma sin tener que intercalar ejecución.

Locks:

Resolver el problema de las race conditions de forma “manual” puede ser altamente

complejo dificultando aún más la programación concurrente. Por esto es que con el fin de simplificar el problema se usan los locks.
Un lock es una variable que permite la sincronización mediante la exclusión mutua (cuando un thread tiene el lock ningún otro puede tenerlo)
Se hace que se requiera tener el lock para poder entrar a cierta parte del código, entonces si lo tiene algún otro no puede entrar hasta que el otro lo libere y el lo agarre. Con esto se puede, entonces, garantizar la atomicidad de las operaciones.

API de locks

lock debe proporcionar un área en la cual cualquier conjunto de instrucciones que se ejecutan ahí tienen garantizada la atomicidad.
Las operaciones del lock son:

- lock posee dos estados **busy** o **free**
- lock inicialmente está siempre en free
- utiliza la primitiva obtener() para pedir el acceso al lock.
    - si el estado es **FREE** entonces el estado pasa a **BUSY**
    - chequear y setear el estado del lock es atómico
    - si un thread adquiere el lock todos los demás threads que quieran entrar chequean si el lock está libre y esperarán allí hasta que se libere y lo agarren.
- lock utiliza la primitiva dejar() poniendo su estado en **FREE** y si hubiera otro esperando para entrar lo deja entrar.

Locks y pthread:

int pthread_mutex_lock (pthread_mutex_t * mutex); // genera un lock

int pthread_mutex_unlock (pthread_mutex_t *mutex); // libera el lock
int pthread_mutex_trylock(pthread_mutex_t * mutex); // devuelve error si el lock solicitado aún no se liberó
int pthread_mutex_timedlock(pthread_mutex_t * mutex, struct timespec *abb_timeout); // si es un timeslice no consigue el mutex, devuelve -1 o 0 si lo bloquea, lo que ocurra antes

Algunas Propiedades:

Un lock debe asegurar:

- **Exclusión mutua:** como mucho un solo Threads posee el lock a la vez.
- **Progress:** Si nadie posee el lock, y alguien lo quiere … alguno debe poder obtenerlo.
- **Bounded waitin**g: Si T quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posible accesos antes que T lo haga.

La **sección crítica** es aquella sección de código que se necesita que se ejecute de forma atómica. Para ello esta sección se encierra en un lock.
Definición Arpachi:
Una **Critical section** es un pedazo de código que accede a una variable compartida y no debe ser ejecutada de forma concurrente por más de un hilo
Exclusión mutua: si un hilo está ejecutando en una critical section, los demas no lo podran hacer

Condition Variables:

Las condition variables permiten a un thread esperar a otro para tomar una acción. Son un objeto sincronizado que permite esperar de forma eficiente esperar por un cambio para compartir algún estado protegido por un lock.
Tiene 3 métodos:

- **wait(Lock *lock):** está llamada en forma atómica:
  - suelta el mutex haciendo unlock
  - suspende la ejecución del Thread que lo llama poniéndolo en la lista de espera de la condition variable;
  - se vuelve a hacer lock del mutex antes de volver del wait.

- **signal()**: toma a un thread de la lista de espera de la condition variable y lo marca como potencialmente seleccionable por el planificador para correr, lo pone en la ready list.
- **broadcast**: este toma a todos los threads de la lista y los marca como seleccionables para correr.

Las condition variables son utilizadas cuando alguna señal tiene que suceder entre los threads (por ejemplo, esperar a un cambio de un estado compartido o variable compartida)
También son llamadas **monitores** y están diseñadas para trabajar en concordancia de los locks. Los monitores garantizan que a lo sumo un thread puede estar ejecutándose dentro del monitor.

Formas de hacer locks:

Para compararlas, se definen 3 criterios:

- cumplen exclusión mutua
- **fairness** (los distintos hilos tienen una chance justo de adquirirlo)
- performance

Controlling Interrupts:
Una de las formas más simples es usando **disable_interrupts**() con esto no se podrá

interrumpir y será atómica efectivamente. Pero están los threads ejecutando una acción privilegiada, cosa no deseable. Además de que un hilo podría pedir el lock y quedarse corriendo para siempre o por mucho tiempo.
Además este método no sirve en multi procesadores ya que puede el de otro procesador ingresar sin más problemas. Por último esta solución tampoco es eficiente.

Test and set: También conocido como atomic exchange. Se usa una variable para definir si el lock está tomado o no. Los problemas son 2:

- Si hay timely interrupts se puede dar la situación de que ambos tengan el flag del lock y por ende ambos entren a las critical section.
- Problema de performance: cuando un thread espera el locke se queda ahí realizando algo llamado spin-waiting lo que es un gasto considerable.

Spin Lock: test and set usando soporte del hardware. Este caso es mutuamente excluyente, pero no es justo ya que se puede generar que un thread esté en spin constante generando starvation. En el aspecto de la performance, en multiprocesadores puede ser bastante efectivo, pero en un monoprocesador no.
Compare and Swap: Otra primitiva de hardware, compara si el valor es lo esperado, si lo es actualiza la memoria apuntada por el valor nuevo. Más poderoso que test y set.
Locks basados en hardware, funcionan y son simples pero son ineficientes por el hecho del spinning. Entonces necesitamos además de soporte de hardware soporte del SO para mejorar nuestro lock.
Yield Approach: cuando llego a un lock ya tomado, dejo la cpu a otro(cambio estado a ready). Luego hacemos un Round Robin entre los threads que tenemos. El problema de esto es que sigue sin ser lo suficientemente eficiente por sí solo y no soluciona el problema del starvation.
Queues: Yield approach pero se usa una cola donde se van poniendo los hilos a ejecutar, con esto hay una forma de controlar quién agarra el lock. Además, tiene un spin lock alrededor del flag. Pero al usar la queue soluciona el problema original del spin lock

A pesar de todo esto **no hay una única forma de los locks**, cada SO lo implementa de una manera, aunque todas tienen sus claras similitudes.

Linux usa un two-phase locks. Primero hay un spin esperando a poder agarrar el lock. Pero si no lo puede agarrar entra en una segunda fase en la que se lo pone a dormir hasta que el lock esté libre.

Errores comunes en concurrencia: (heisenbug)

Dos tipos de errores:

- Non-deadlock Bugs
- Deadlock-Bugs

Non-Deadlock bugs:

- Atomicity Violation: en un bloque que debería ser atómico otro thread ejecuta algo entre 2 instrucciones
- Order Violation: Se modifica el orden deseado de acceso a memoria

Deadlock Bugs:

Entre 2 o más threads uno obtiene el lock y por algún motivo no lo libera haciendo que se bloqueen los demás.
Dahlin: Un deadlock es un ciclo de espera a través de un conjunto de threads en el cual cada thread espera que algún otro thread en el ciclo tome alguna acción.

DeadLock: consideraciones necesarias

- Exclusión mutua: los thread reclama control exclusivo sobre un recurso compartido que necesitan.
- Hold-and-Wait: un thread mantiene un recurso reservado para sí mismo mientras espera que se de alguna condición.
- No preemption: los recursos adquiridos no pueden ser desalojados (preempted) por la fuerza.
- Circular wait: existe una conjunto de threads que de forma circular cada uno reserva uno o más recursos que son requeridos por el siguiente en la cadena.

Prevención de Deadlock:

Circular Wait: Escribir el código de lockeo de forma tal que nunca se de una circular wait. Para esto se provee un ordenamiento de cómo se adquieren los locks.
Hold and wait: Este se puede resolver adquiriendo todos los locks de una, de manera atómica. Esta solución complica el encapsulamiento, ya que es necesario saber para cada función exactamente que locks necesita tener y adquirirlos antes de empezar. Además hará al programa menos concurrente.
No preemption: Buscar aprovechar la API para desbloquear un lock y poder seguir evitando el problema. Esto puede generar un livelock en el q 2 hilos quieren liberar un lock para agarrarlo y por eso ninguno puede tenerlo. Otro problema es con la modularización ya que al poder detenerlo hay que ser muy cuidadoso para terminar la ejecución del otro de forma correcta liberando sus recursos.
Exclusión mutua: La solución de buscar evitar estas secciones es compleja, ya que iría un poco en contra de la idea de concurrencia. Para esto, una posibilidad es el uso de poderosas instrucciones de hardware construyendo estructuras que no requieran del lock explícito.

En algunos casos es **preferible evitar los deadlocks**. Para esto, hay **smart**

schedulers los cuales pueden **elegir qué hilo correr** según lo que estarían haciendo y con eso evitar deadlocks

Detect and recover:

Otra estrategia es permitir ocasionalmente la ocurrencia de deadlocks para detectarlos y luego tomar acción sobre estos. Por ejemplo hay casos de deadlocks extrañas que ocurren poco por probabilidad, por ejemplo una vez al año. Entonces por la complejidad que tendría buscar la solución y su ocurrencia es preferible que cuando suceda se reinicie y listo antes que buscar la solución. Esto está implementado incluso en algunos motores de bases de datos que cuando detectan que cayeron en un deadlock reinician y siguen como si nada hubiese pasado.