El Kernel

Ejecución Directa

Correr el programa directamente en la CPU.
Ventajas:

- Rapidez

Desventajas:

- El SO no puede asegurarse que el programa no va a hacer nada que el usuario no quiera
- El SO no puede pausar su ejecución y hacer que otro sea ejecutado

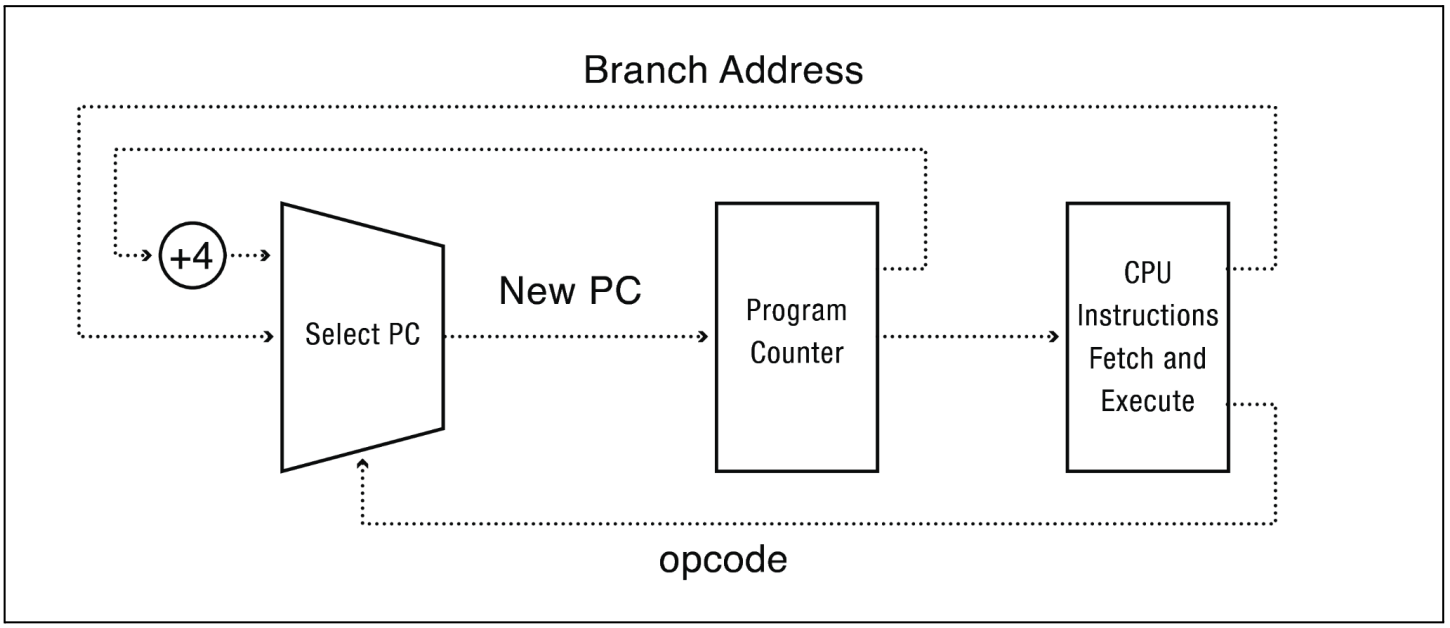
Limitar la Ejecución Directa

Se realiza por distintos mecanismos de hardware:

- Dual Mode Operation
- Privileged Instructions
- Memory Protection
- Timer Interrupts

Modo Dual de Operaciones

Mecanismo que proveen todos los procesadores y algunos microprocesadores modernos. Idealmente, un esquema básico sería:

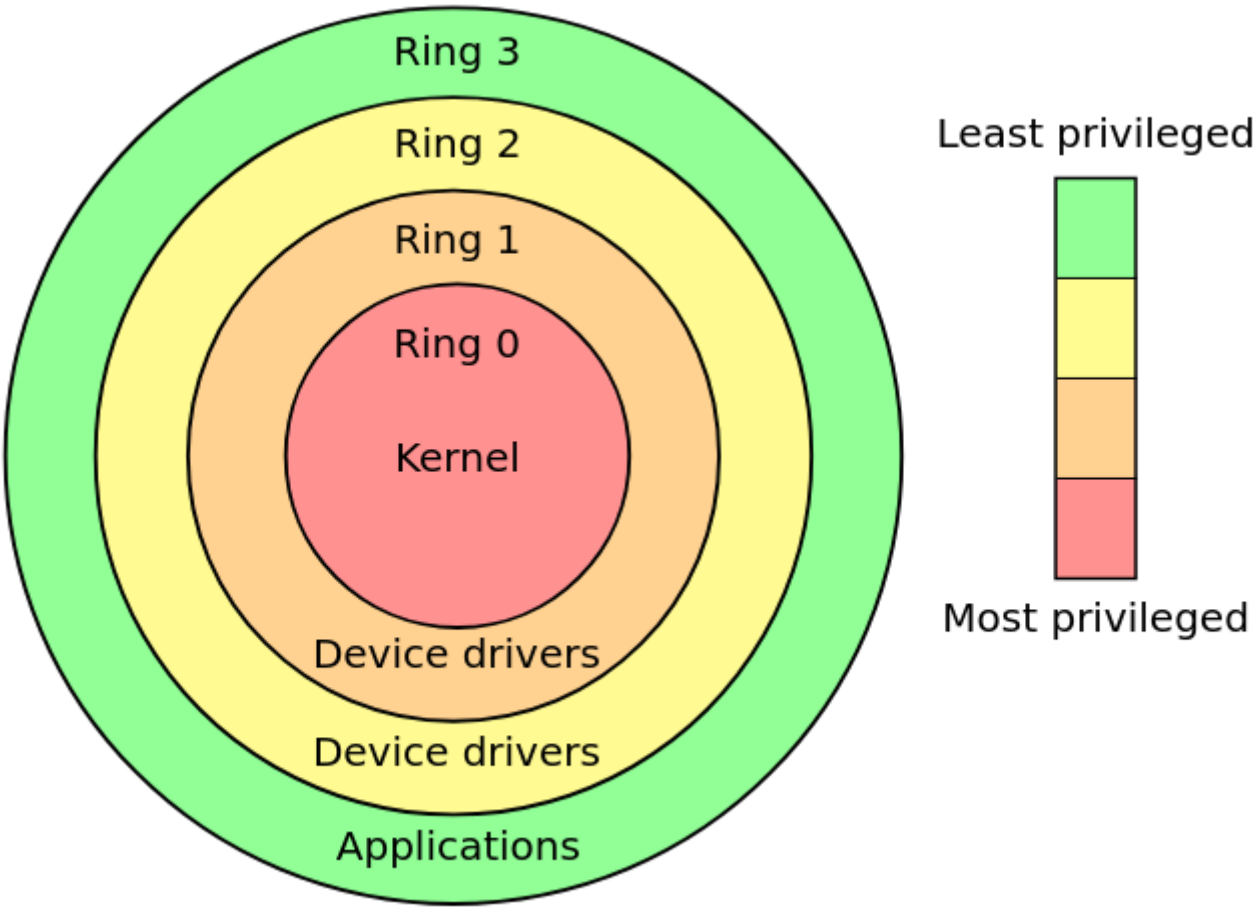


Kernel Land & User Land

Los Modos

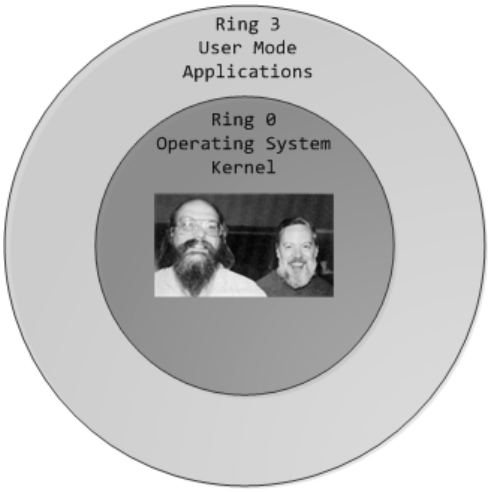
Para la arquitectura x86 existen 4 modos de operaciones vía hardware denominados **rings** y están numerados entre 0 y 3.

Los modos más utilizados por los SO son el modo 0 (supervisor/Kernel) y el modo 3 (usuario).
Al ser un mecanismo proveído por el hardware, cada instrucción a ser ejecutada es chequeada previamente por el mismo para identificar en qué modo de operación se encuentra.



Los modos operacionales utilizados de un procesador son:

- **Modo Usuario / User Mode**: ejecuta instrucciones en nombre del usuario
- **Modo Supervisor / Kernel / Monitor**: ejecuta instrucciones en nombre del Kernel y son privilegiadas



Estos equivalen a un bit en el registro de control del procesador (instrucciones privilegiadas // normales). A cada uno de estos niveles de privilegio también se lo denominan Ring o Anillo.
Lo que se busca es proteger el hardware (memoria, puertos de I/O y posibilidad de ejecutar ciertas instrucciones).

IOPL

Indicador que se encuentra en la CPU en modos: protegido y largo. Cuando el nivel de privilegio actual (CPL: CPL0, CPL1, CPL2, CPL3) del programa o tarea actual sea menor o igual que el IOPL, el/la mismo/a tendrá acceso a los puertos de E/S. Se encuentra en todas las CPU x86 compatibles con IA-32 y ocupa los bits 12 y 13.

Se puede cambiar sólo cuando el nivel de privilegio actual es Ring 0.
Además de IOPL, los permisos de puerto de E/S en el Task State Segment también participan en la determinación de la capacidad de una tarea para acceder a un puerto de E/S.

Privileged Instructions

Por la existencia del Modo Dual, los distintos modos poseen cada uno su propio set de instrucciones que pueden ser ejecutadas o no según el bit de modo de operación.

Memory Protection

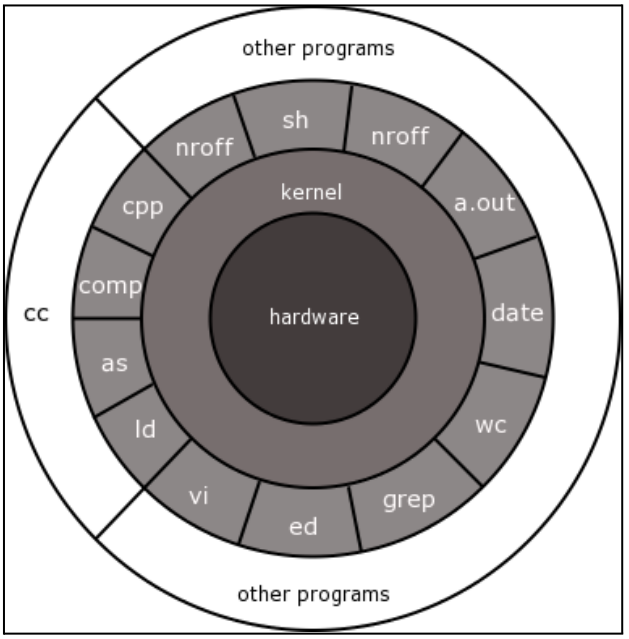
El SO y los programas que están siendo ejecutados deben residir ambos en memoria al mismo tiempo (el SO debe cargar el programa y hacer que comience a ejecutarse y el programa tiene que residir en memoria para poder hacerlo), de modo que -para que la memoria sea compartida de forma segura- el SO debe poder configurar el hardware de forma tal que cada proceso pueda leer y escribir su propia porción de memoria.

Timer Interrupts

Permite que el procesador sea interrumpido después de un determinado lapso de tiempo. Cada procesador tiene su propio contador de tiempo. Si el procesador es multi-core, cada core tiene sus propios contadores de tiempo individual.
Es un mecanismo que periódicamente le permite al Kernel desalojar al proceso de usuario en ejecución y volver a tomar el control del procesador -y así de toda la máquina-.
El reseteo del timer es una instrucción privilegiada que puede ser utilizada en modo Kernel.

Definición

Es la capa de software (y por tanto un programa en sí mismo) de más bajo nivel en la computadora. Controla los recursos de hardware de una computadora y provee un ambiente bajo en el cual los programas puede ejecutarse.
Los programas son independientes del hardware y es fácil moverlos entre sistemas UNIX que se ejecutan en hardware diferente dado que los programas no hacen suposiciones sobre el mismo, si no que se comunican con el Kernel.
Cuando el código fuente de esta capa es ejecutado, la computadora pasa al estado **Supervisor**.



Tareas Específicas

- Planificar la ejecución de las aplicaciones
- Gestionar la memoria
- Proveer un sistema de archivos
- Creación y finalización de procesos
- Acceder a los dispositivos
- Comunicaciones
- Proveer una API

Modos de Transferencia

Formas de alternar entre modo usuario y modo Kernel.

De Modo Usuario a Modo Kernel

- Interrupciones -> por un evento externo
- Excepciones del procesador -> por un evento interno (de hardware) inesperado causado por un programa de usuario
- Ejecución de system calls -> por un evento intencional generado por una system call (int 0x86)

Interrupciones

Señal asincrónica enviada hacia el procesador de que algún evento externo ha sucedido y pueda requerir de la atención del mismo.
El procesador está continuamente chequeando si una interrupción es disparada.
Cuando se dispara una interrupción, el procesador completa o detiene la instrucción que se esté ejecutando, guarda todo el contexto y comienza a ejecutar el manejador de esa interrupción en el Kernel.
El orden de prioridad de las interrupciones (según BCH) es:

1. Errores de la máquina
2. Timers
3. Discos
4. Network devices
5. Terminales
6. Interrupciones de software

Excepciones del procesador

El funcionamiento es igual al de una interrupción.
Podrían ser: acceder fuera de la memoria del proceso, intentar ejecutar una instrucción privilegiada en modo usuario, intentar escribir en memoria de sólo lectura, dividir por cero.

Ejecución de system calls

Las System Calls son funciones (expuestas por el Kernel) que permiten a los procesos de usuario pedirle al Kernel que realice operaciones en su nombre.
Las System Calls conforman una API.

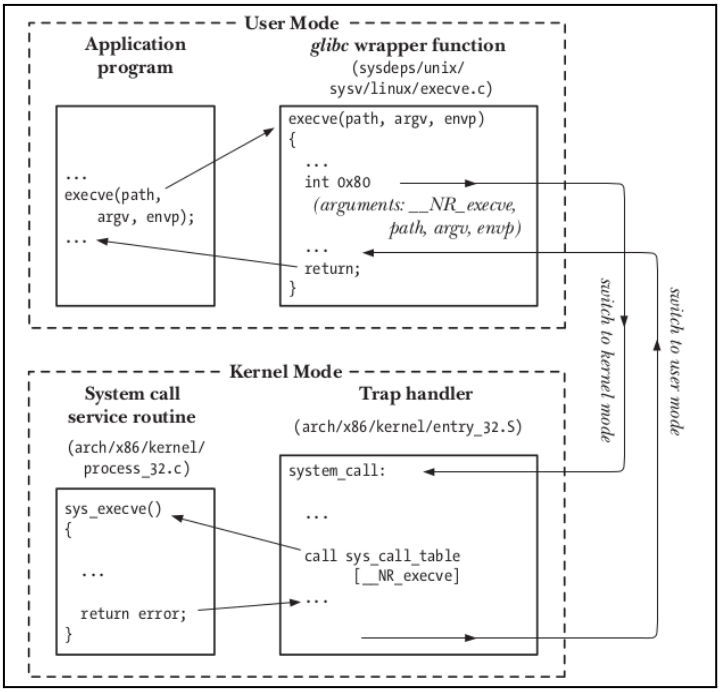
System Call (syscall)

Punto de entrada controlado al Kernel que permite a un proceso solicitarle realizar alguna operación en su nombre. El Kernel expone una gran cantidad de servicios accesibles por un programa vía el API (application programming interface) de syscalls.

Características:

- Una syscall cambia el modo del procesador de user mode a Kernel mode de modo que la CPU podrá acceder al área protegida del Kernel.
- El conjunto de syscall es fijo, y cada una está identificada por un único número (que no es visible al programa, quien sólo conoce su nombre).
- Cada syscall debe tener un conjunto de parámetros que especifican información que debe ser transferida desde el user space al Kernel space.

Llamada a una syscall



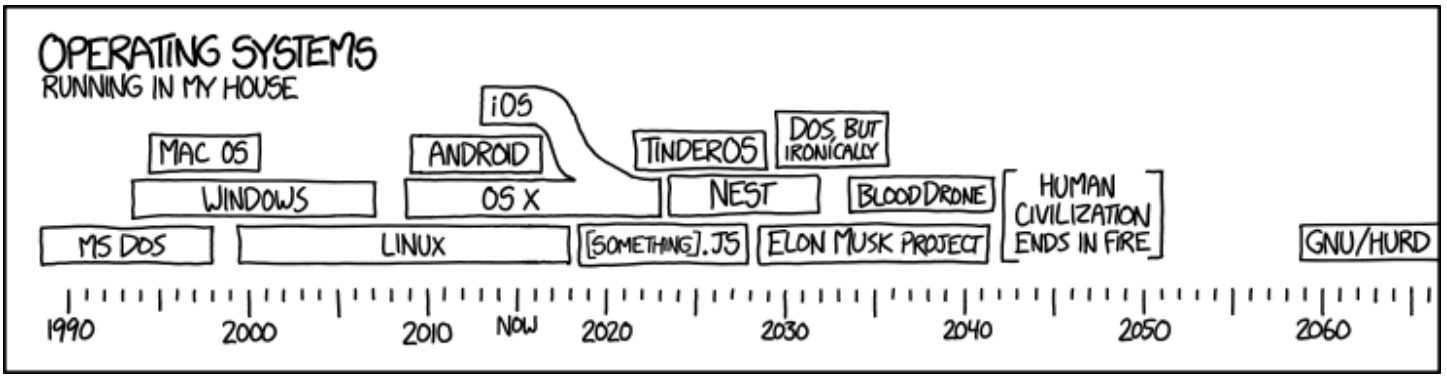
1. El programa llama a una syscall mediante un wrapper en la biblioteca de C.
2. El wrapper proporciona todos los argumentos a la syscall trap_handling. Los argumentos son pasados al wrapper por el stack, pero el Kernel los espera en determinados registros. El wrapper copia los valores a los registros.
3. El wrapper copia el número de la syscall a un determinado registro de la CPU (%eax) para que el Kernel pueda identificarla.
4. El wrapper ejecuta una instrucción de código máquina llamada trap machine instruction (int0x80) que causa que el procesador pase de user mode a Kernel mode y ejecute el código apuntado por la dirección 0x0 (128) del vector de traps del sistema.
5. En respuesta al trap de la posición 128, el Kernel invoca su propia función ***system_call()*** para manejar esa trap. El manejador:
   a. graba el valor de los registros en el stack del Kernel
   b. verifica la validez del número de la syscall
   c. invoca el servicio correspondiente a la syscall llamada a través del vector syscalls, realiza su tarea y le devuelve un resultado de estado a la rutina syscall
   d. se restauran los registros almacenados en el stack del Kernel y se agrega el valor de retorno en el stack
   e. se devuelve el control al wrapper y se pasa a user mode
6. Si el valor de retorno de la rutina de servicio de la syscall da error, el wrapper setea el valor en errno.

Pasar de Kernel Mode a User Mode

- Un **nuevo proceso**:
  a. el Kernel copia el programa en la memoria
  b. setea el contador de programa apuntando a la primera instrucción del proceso
  c. setea el stack pointer a la base del stack de usuario
  d. switchea a modo usuario
- Continuar después de una **interrupción**, una **excepción del procesador** o de una **syscall**: una vez que el Kernel terminó de manejar el pedido, continúa con la ejecución del proceso interrumpido mediante la restauración de todos los registros y cambiando el modo a nivel usuario.
- Cambio entre **diferentes procesos**: el Kernel decide ejecutar otro proceso que no sea el que estaba ejecutando y carga el estado del nuevo a través de la PCB y cambia a modo usuario.

El Sistema Operativo UNIX y sus Distintos Sabores

Por ser una familia de sistemas operativos por lo que no existe un único Kernel:



UNIX

- 1969: Ken Thompson utilizó una PDP-7 que se encontraba en desuso para implementar una versión mínima del SO llamado MULTICS.
- 1971 - UNIX-v1: Primera versión de UNIX fue lanzada como proyecto en los laboratorios Bell, por Dennis Ritchie y Ken Thompson, escrito en assembler con:
  - 16 Kb para el SO,
  - 8 Kb para programas de usuario,
  - 512 Kb de almacenamiento
  - 64 kb de límite por archivo
- 1972 - UNIX-v2
- 1973 - UNIX-v3: Versión que tenía un compilador en C.
- 1973 - UNIX-v4: Primera versión escrita totalmente en C.
- 1974 - UNIX-v5
- 1975 - UNIX-v6: Versión (tal vez) más conocida por su licencia gratuita para investigación y enseñanza.
- 1979 - UNIX-v7: Versión padre de todos los SO basados en UNIX (excepto Coherent, Minix y Linux).

Linux

Kernel desarrollado por Linus. No es un SO sino que el resto de los componentes son desarrollados por terceros como GNU.
Características:

- El Kernel no tiene acceso a la biblioteca de C ni a los encabezados C estándar
- El Kernel está codificado en GNU C
- El Kernel carece de la protección de memoria internamente (no se protege de sí mismo)
- El Kernel no puede ejecutar fácilmente operaciones de punto flotante
- El Kernel tiene una pequeña pila de tamaño fijo y no es dinámica
- El Kernel es preventive y admite SMP por las interrupciones sincrónicas (la sincronización y la concurrencia son preocupaciones importantes)
- La portabilidad es importante

El Kernel de Linux se denomina **monolítico**, pues es el único programa ejecutándose en la memoria del computador.

MINIX

SO de la familia de UNIX, creado por Andrew S. Tanenbaum en Amsterdam.
El Kernel utiliza una filosofía distinta que el de Linux o de un UNIX tradicional, pues se basa en el concepto de **microkernel** pequeño que maneja interrupciones, gestión de procesos de bajo nivel e IPC.
El SO es de tipo **multiservidor** pues se ejecuta como una colección de controladores y servidores en modo de usuario con modos de falla independientes mientras que el microkernel se ejecuta en modo Kernel.

BSD

Tipos de Kernel

Existen dos tipos de estructuras:

- **Kernel Monolítico**: el Kernel es un único 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, mientras que otros servicios que nos son imprescindibles que se ejecuten en modo Kernel se implementan en Ring 1 o 2.

Iniciar al Sistema Operativo y el Kernel

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

1. **Booteo**: proceso denominado bootstrap, que depende del hardware de la computadora. Se realizan los chequeos de hardware y se carga el bootloader que es el programa encargado de cargar el Kernel del SO.
    a. Cargar el BIOS (Basic Input/Output System)

    b. Crear la Interrupt Vector Table y cargar las rutinas de manejo de interrupciones en Modo Real
    c. La BIOS genera una *interrupción 19* (INT 19)
    d. Se ejecuta el servicio de interrupciones y el handler de dicha interrupción dispara la lectura al primer sector de 512 bytes del disco a memoria
2. **Carga del Kernel**: el BootLoader se encarga de:
    a. pasar a modo supervisor (pues se realiza por hardware)
    b. ir a buscar el Kernel al dispositivo donde se encuentra almacenado
    c. carga el Kernel en memoria principal
    d. setea el registro de PI (próxima instrucción)
    e. ejecuta la primer instrucción del Kernel
3. **Inicio de las Aplicaciones de Usuarios**: al finalizar la ejecución del Kernel, las últimas operaciones que realiza son:
    a. cargar en memoria la aplicación que debe ejecutar (usualmente la shell)
    b. setear el PI en la primera instrucción de esta aplicación
    c. pasar a modo usuario y dejar el control a la aplicación

Fase de Carga del Kernel

El Kernel es cargado como un archivo de imagen, comprimido dentro de otro. Contiene una cabecera de programa que hace una cantidad mínima de instalación del hardware y descomprime la imagen completamente en la memoria alta. Finalmente, lleva a cabo su ejecución llamando la función startup del Kernel.

Fase de Inicio del Kernel

La función de arranque para el Kernel (proceso 0):

- establece la gestión de memoria
- detecta el tipo de CPU
- cambia las funcionalidades del Kernel para arquitectura no específicas de Linux, a través de la función start_kernel()

A su vez, start_kernel():

- establece el manejo de interrupciones (IRQ)
- configura memoria adicional
- comienza el proceso de inicialización

En particular, se monta el disco RAM inicial que se ha cargado anteriormente como el sistema raíz temporal durante la fase de arranque, por lo que, los módulos controladores se cargan sin depender de otros dispositivos físicos y drivers y mantiene el Kernel más pequeño.
El sistema de archivos raíz es luego cambiado a través de la función pivot_root(), que desmonta el sistema de archivos temporal y lo reemplaza por el real una vez que éste sea accesible y la memoria temporal es recuperada.

El Proceso de Inicio

Una vez que el Kernel está totalmente en funcionamiento, Init establece y opera todo el espacio de usuario, tal como:

- la comprobación y montaje de sistemas 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

Durante el arranque del sistema, se verifica si existe un nivel de ejecución predeterminado, si no, se debe introducir por medio de la consola del sistema. Después se procede a ejecutar todos los scripts relativos al nivel de ejecución especificado.

Finalmente, Init se aletarga y espera a que uno de estos tres eventos sucedan:

➔ que procesos comenzados finalicen o mueran,
➔ un fallo de la señal de potencia (energía),
➔ o una petición a través de /sbin/telinit para cambiar de ejecución.

El Proceso

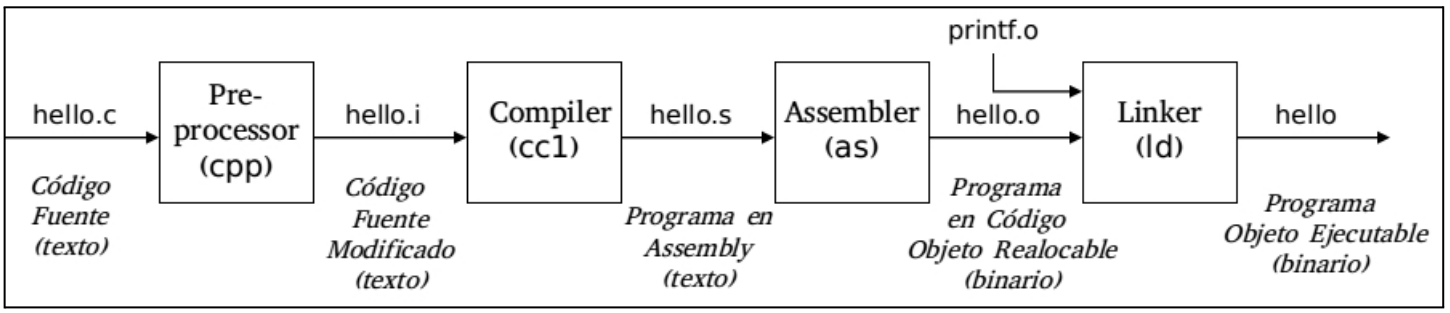
De Programa a Proceso

Una vez que se edita un programa en un lenguaje de programación, se compila para poder obtener un programa ejecutable.
Un programa es algo sin vida: un conjunto de instrucciones y datos que esperan en algún lugar del disco para saltar a la acción. El SO toma ese puñado de bytes y lo transforma en algo útil mediante el Kernel.
El Kernel se encarga de:

1. cargar instrucciones y datos de un programa ejecutable en memoria
2. crear el stack y el heap
3. transferir el control al programa
4. proteger al SO y al programa

Fases de la Compilación

1. **Procesamiento**: El preprocesador (cpp) modifica el código de fuente original de un programa escrito en C y devuelve otro programa con extensión *.i*
2. **Compilación**: El compilador (cc) traduce el programa *.i* a un archivo de texto *.s* que contiene un programa en lenguaje assembly
3. **Ensamblaje**: El ensamblador (as) traduce el archivo *.s* en instrucciones de lenguaje de máquina empaquetándolas en un programa objeto realocable, el cual es almacenado en un archivo con extensión *.o*
4. **Link-Edición**: El linker (ld) mezcla las funciones provistas por la biblioteca de C que fueron utilizadas en el código fuente, teniendo como resultado un archivo objeto ejecutable



Algunas particularidades que ocurren durante de la compilación son:

- ➔ **Instrucciones de Lenguaje de Máquina**: almacena el código del algoritmo del programa
- ➔ **Dirección del Punto de Entrada del Programa**: identifica la dirección de la instrucción con la cual la ejecución del programa debe iniciar
- ➔ **Datos**: el programa contiene valores de los datos con los cuales se deben inicializar variables, valores de constantes y literales utilizadas
- ➔ **Símbolos y Tablas de Realocación**: describe la ubicación y los nombres de las funciones y variables de todo el programa
- ➔ **Bibliotecas Compartidas**: describe los nombres de las bibliotecas compartidas que son utilizadas por el programa en tiempo de ejecución así como también

la ruta del linker dinámico que debe ser utilizado para cargar la biblioteca en cuestión

➔ **Otra información**: el programa contiene información adicional necesaria para terminar de construir el proceso en memoria

Un Programa en Unix

Un programa contiene un **formato de identificación binaria**, esto es: cada archivo ejecutable posee META información describiendo el formato ejecutable, lo que al Kernel interpretar la información contenida.
Algunos formatos en Unix son:

- OUT: Assembler Output -> Salida del compilador de C
- COFF: Common Object File Format -> Utilizado en las versiones de System V
- ELF: Executable and Linking Format -> Utilizado en la actualidad

Definición

Un Proceso es un programa en ejecución que incluye:

- archivos abiertos
- señales pendientes
- datos internos del Kernel
- estado completo del procesador
- espacio de direcciones de memoria
- uno o más hilos de ejecución, cada uno con:
  - un único program counter
  - un stack
  - un conjunto de registros
- una sección de datos globales

Además, el Kernel en sí también es un proceso.

La virtualización

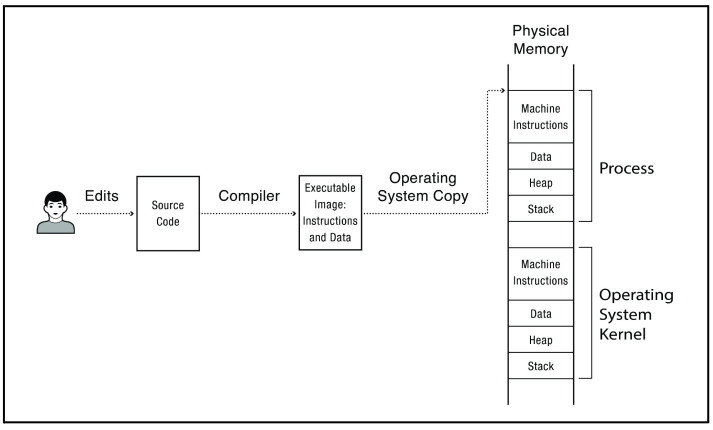
Existen dos tipos de virtualización en los sistemas operativos modernos:

★ De Memoria
★ De Procesador

De Memoria

Hace creer al proceso que tiene toda la memoria disponible para ser reservada y usada como si estuviera siendo ejecutado sólo en la computadora.
Todos los procesos en Linux están divididos en 4 segmentos:

- Text: instrucciones del programa
- Data: variables globales (extern o static en C)
- Heap: memoria dinámica alocable



- Stack: variables locales y trace de llamadas

Estas secciones se denominan: **espacio de direcciones** del proceso.
Para ejecutar un programa, el SO:

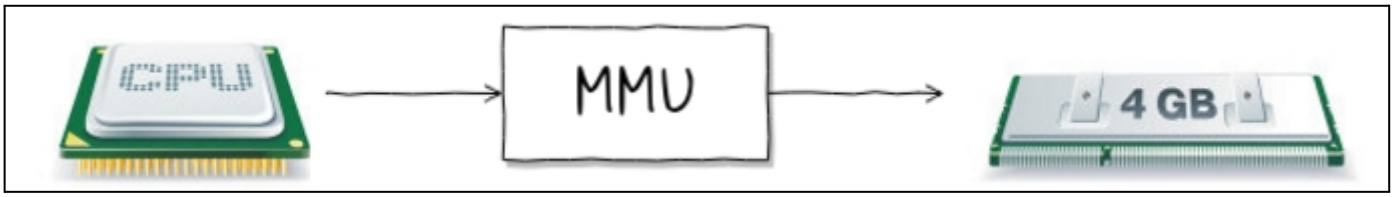
1. Copia las instrucciones en la sección .code y los datos en la sección .data, desde el programa ejecutable residente en disco hacia la memoria física
2. Setea el .stack (región de memoria, aka: execution stack) que mantiene el estado de las variables locales durante las llamadas a los procedimientos.
3. Setea el .heap (otra región de memoria) destinada a alojar cualquier estructura de datos alocada en forma dinámica que el programa pueda necesitar.

Protección de Memoria - Memoria Virtual

Abstracción por la cual la memoria física puede ser compartida por diversos procesos.
Se manejan las llamadas **direcciones virtuales**: direcciones emitidas por la CPU, permiten que cada proceso inicie su memoria en el mismo lugar (dirección 0). El hardware traduce la dirección virtual a una **dirección física** de memoria.

Traducción de Direcciones - Dirección Virtual a Dirección Física

Mapeo que se realiza por hardware en la **MMU**: Memory Management Unit.



De Procesador

Virtualización de Procesamiento: forma de virtualización más primitiva que consiste en dar la ilusión de la existencia de un único procesador para cualquier programa que requiera de su uso, proveyendo:

- 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 potentes
- Aislamiento frente a fallas:
  - los procesos no pueden directamente afectar a otros
  - los errores no colapsan toda la máquina

EL SO crea esta ilusión mediante la virtualización de la CPU a través del Kernel

PCB: Process Control Block

Estructura que le permite al SO

- llevar una contabilidad de todos los procesos que se estén ejecutando
- almacenar toda la información que debe conocer sobre un proceso: dónde se encuentra almacenado, dónde se encuentra almacenada su imagen ejecutable, qué usuario lo solicitó, qué privilegios tiene

Por dentro

Un proceso necesita permisos del Kernel del SO para:
- acceder a memoria perteneciente a otro proceso
- escribir/leer en el disco
- cambiar algún seteo del hardware del equipo
- enviar información a otro proceso

Arquitectura de Von Newman: Instruction Fetch

El ciclo de una instrucción en una arquitectura VN es:
1. Fetch: obtener la instrucción
2. Decode: decodificar la instrucción
3. Execute: ejecutar la instrucción
4. CP: próxima instrucción

El API de Procesos

Cualquier interfaz de un SO debe incluir:
- ❖ Create: crear un nuevo proceso
- ❖ Destroy: destruir por la fuerza un proceso
- ❖ Wait: esperar a que un proceso termine su ejecución
- ❖ Miscellaneous Control: operar con el proceso (ej: suspender su ejecución por un tiempo y luego reanudarla)
- ❖ Status: establecer una situación particular a un proceso y su información relacionada (ej: cuánto hace que se está ejecutando)

El API de Procesos de Unix-like: System Calls

Un proceso en Unix sólo puede ser creado por otro: el creador se denomina **padre** y el creado **hijo**.

El Contexto de un Proceso

Según [VAH]

Cada proceso tiene un contexto que comprende la información para describirlo:
- ➔ **User Address Space**: usualmente dividido en text, data, stack y heap.
- ➔ **Control Information**: se utilizan dos estructuras para mantener información de control de un proceso (*u area* y *proc*).
- ➔ **Credentials**: credenciales que incluyen los grupos IDs y UserId asociados.
- ➔ **Variables de Entorno**: conjunto de valores heredados del proceso padre.
- ➔ **Hardware Context**: contenido de los registros de propósito general y de un conjunto especial de registros del system:
  - ◆ PC -> program counter
  - ◆ SP -> stack pointer
  - ◆ PWD -> processor status word
  - ◆ memory management registers
  - ◆ registros de la unidad de punto flotante

Según [BHC]

El contexto de un proceso consiste en la unión de user-level context, register context y system level context:

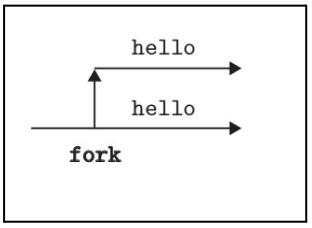
- ➔ **User-level Context**: secciones que forman parte de Virtual Address Space del proceso
  - ◆ text
  - ◆ data
  - ◆ stack
  - ◆ heap
- ➔ **Register Context**: contador de Programa
  - ◆ registro del estado del procesador (PS)
  - ◆ stack pointer
  - ◆ registros de propósito general
- ➔ **System-level Context**:
  - ◆ entrada en la Process Table entry
  - ◆ u area
  - ◆ process region entry, region table y page table que definen el mapeo de la memoria virtual vs memoria física del proceso

U Area y Estructura Proc

- U Area (user area):
  - Parte del espacio del proceso
  - Se mapea y es visible por el proceso sólo cuando este está siendo ejecutado
  - Almacena información que sólo es necesario acceder cuando el proceso se esté ejecutando
  - Contiene
    - Process Control Block (guarda el hardware context cuando el proceso no se está ejecutando)
    - Puntero a la Proc Structure del proceso
    - UID y GID real
    - Argumentos para y valores de retorno o errores hacia la system call actual
    - Manejadores de Señales
    - Información sobre las área de memoria (text, data, stack, heap)
    - Tabla de descriptores de archivos abiertos (Open File descriptor Table)
    - Puntero al directorio actual
    - Datos estadísticos del uso de la CPU, información de perfilado, uso de disco y límites de recursos
- Proc Structure (aka: proc structure in linux):
  - Entrada en la *Process Table*
  - Almacena información necesaria incluso cuando el proceso no se está ejecutando
  - Contiene

- Identificación (cada proceso tiene un identificador único o process ID y además pertenece a un determinado grupo de procesos)
- Ubicación del mapa de direcciones del Kernel del u area del proceso
- Estado actual del proceso
- Puntero hacia el siguiente proceso en el planificador y al anterior
- Prioridad
- Información para el manejo de señales
- Información para la administración de memoria

fork()



Syscall

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

Algoritmo

- 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
- marcar 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 con datos para que el hijo se reconozca a sí mismo y para que tenga un punto de inicio cuando el planificador lo haga ejecutarse

Curiosidades:

- ➢ Una llamada y dos valores de retorno
- ➢ Ejecución concurrente
- ➢ Address Space duplicados pero separados (el address space de cada proceso son idénticos)
- ➢ Archivos compartidos (la user file descriptor table es heredada con todos sus archivos en el mismo estado)

Errores: se devuelve

- ➢ Valor 0 al proceso hijo
- ➢ El PID del hijo al proceso padre
- ➢ Un valor negativo

exit()

Syscall: _exit()

Un proceso tiene dos formas de terminar:

- Anormal: a través de recibir una señal cuya acción por defecto es terminar el programa
- Normal: a través de invocar a la syscall exit()

Algoritmo

- ignorar todas las signas
- cerrar todos los archivos abiertos
- liberar todos los locks mantenidos por el proceso sobre los archivos
- liberar el directorio actual
- separar los segmentos de memoria compartida
- actualizar los contadores de los semáforos
- liberar todas las secciones y memoria asociada al proceso
- registrar información sobre el proceso
- poner el estado del proceso en *zombie*
- asignar el pid de los procesos hijos al pid de *init*
- enviar una señal de muerte al proceso padre
- context switch

wait()

Sincroniza la ejecución de un proceso con la finalización de un proceso hijo retrasando la ejecución del proceso padre hasta que el hijo termine su ejecución.

exec()

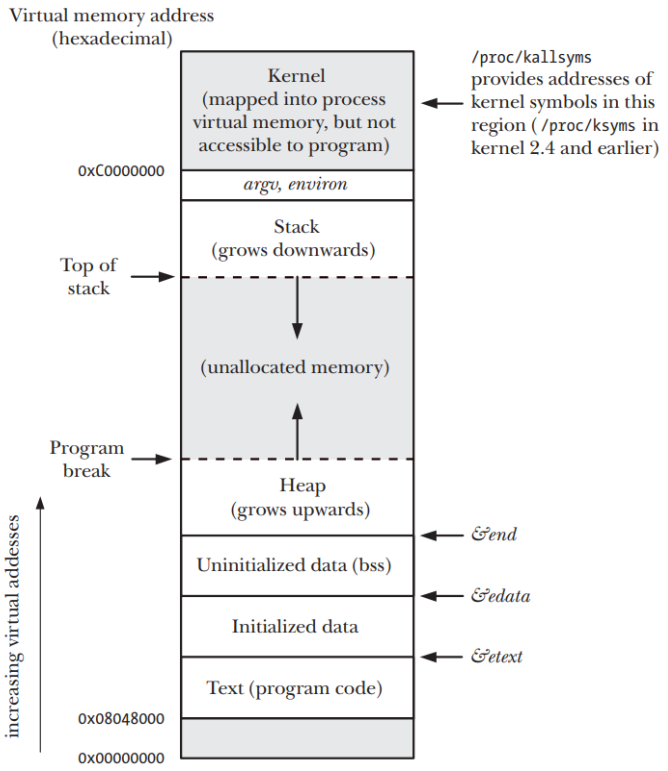
Syscall

Invoca a otro programa, sobreponiendo el espacio de memoria del proceso con el programa ejecutable.

Algoritmo

- obtener el inodo del programa
- verificar si el archivo es ejecutable y si el usuario tiene los permisos para ejecutarlo
- leer el header del archivo
- copiar los parámetros del exec del viejo address space al system space
- des-asociar cad región asociada al proceso
- alocar espacio para la nueva región
- asociar la nueva región
- cargar la nueva región en memoria
- copiar los parámetros del exec en la nueva región o sección stack
- inicializar a modo usuario
- liberar el inodo

brk()



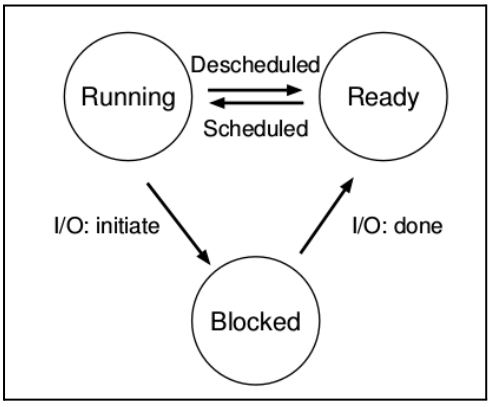
El final del heap se denomina break: aumenta hacia direcciones altas (grows upwards) y el stack se crece hacia direcciones bajas (grows downwards).
Un proceso puede reservar memoria para sí mismo incrementando el tamaño del heap utilizando la familia de funciones malloc(), la cual está basada en la syscall brk().
Inicialmente, el break del programa está ubicado en el final de datos no inicializados. Después que brk() se ejecuta, el break es incrementando y el proceso puede acceder a cualquier memoria en la nueva área reservada, pero no accede directamente a la memoria física.
El parámetro de sbrk() es la dirección exacta donde el nuevo break debe estar y se pasa el incremento al cual se le sumará al viejo break para setear el nuevo. Si se ejecuta sbrk(0) se obtiene la dirección del break actual.

Metamorfosis: De Programa a Proceso

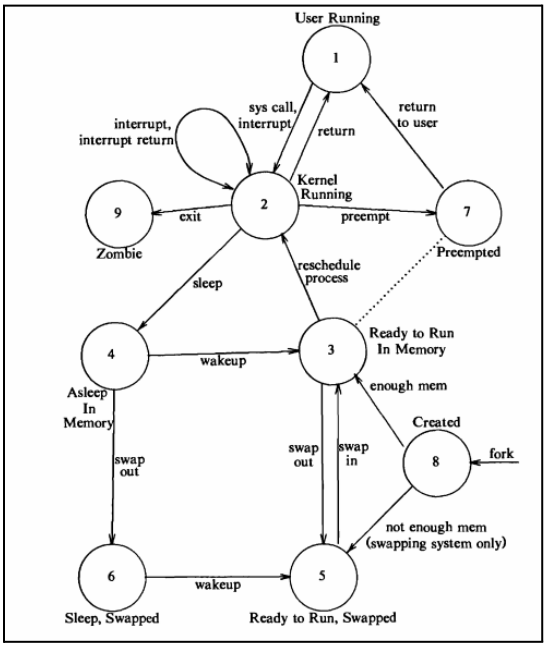
1. El SO debe cargar el programa, su código y cualquier dato estático en la memoria. Los programas residen en disco en algún formato ejecutable (en linux: elf). Se realiza de forma perezosa (lazily) cargando lo que se necesite según se necesite.
2. Se crea la pila de ejecución (execution stack) en base a reservar cierta cantidad de memoria y se inicializa si es necesario (ejemplo: con argv y argc del main()).
3. Se crea el heap en base a reservar otra cierta cantidad de memoria, pero esta de tipo dinámica (en C se crea y se destruyen estructuras de memoria dinámica con malloc() y free()).
4. El SO deberá realizar otras operaciones (varias relacionadas con entrada y salida de datos).
5. Se setea el punto de entrada (entry point) de ejecución de las instrucciones del programa en el main().

Estados de un Proceso

- ★ Running -> ejecutando instrucciones en un procesador
- ★ Ready -> listo para correr
- ★ Blocked -> ejecución en pausa hasta que algún evento suceda



Estados en Unix System V

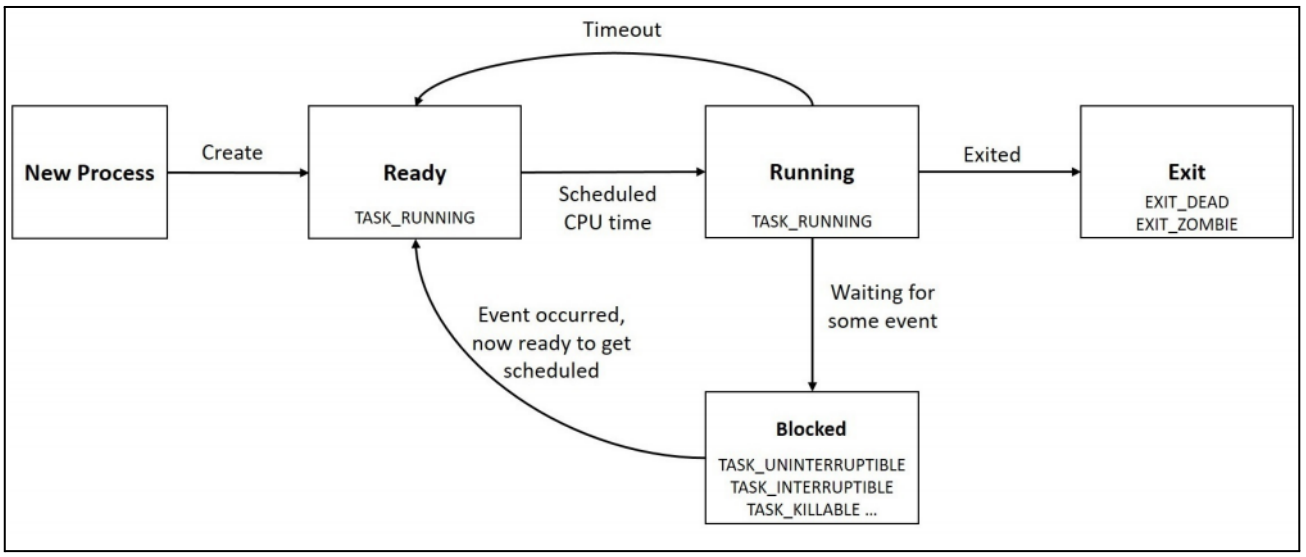


- ➔ **Running User Mode**
- ➔ **Running Kernel Mode**
- ➔ **Ready to Run on Memory**
- ➔ **Asleep in Memory** -> bloqueado en memoria
- ➔ **Ready to Run but Swapped** -> listo para correr en memoria secundaria
- ➔ **Asleep Swapped** -> bloqueado en memoria secundaria
- ➔ **Preempt** -> running user mode que pasó antes por Kernel mode
- ➔ **Created** -> recién creado en un estado de transición
- ➔ **Zombie** -> el proceso padre ejecutó la syscall exit() y no existe más

Estados en Linux

Como en Linux los procesos son denominados tasks, los estados pueden ser:

- ➔ **TASK_RUNNING (0)** -> ejecutándose o esperando CPU en la cola de run del scheduler
- ➔ **TASK_INTERRUPTIBLE (1)** -> espera hasta que la condición que lo pausó sea verdadera (cualquier señal generada para el proceso es entregada al mismo)
- ➔ **TASK_KILLABLE** -> similar a TASK_INTERRUPTIBLE con la excepción que las interrupciones pueden ocurrir en fatal signals
- ➔ **TASK_UNINTERRUTPIBLE (2)** -> ininterrupción similar al anterior pero no podrá ser despertado por las señales que le lleguen
- ➔ **TASK_STOPPED (4)** -> recibió una señal de STOP y volverá a running cuando reciba la señal para continuar (SIGCONT)
- ➔ **TASK_TRACED (8)** -> está siendo revisado (probablemente por un debbuger)
- ➔ **EXIT_ZOMBIE (32)** -> 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



Desde el Kernel de Linux

Task List: lista circular doblemente enlazada donde el Kernel almacena la lista de procesos. Cada elemento es un descriptor del proceso (process descriptor) del tipo *task_struct*.
Los procesos se identifican en Linux (al igual que en Unix) vía el PID.

Scheduling - Planificación de Procesos

Consta de determinar cuánto tiempo de CPU le toca a cada proceso y sucede en los SO de tipo *time sharing*. El período de tiempo que el Kernel le otorga a un proceso se denomina **time slice** o **time quantum**.

Multiprogramación

Utilización de la CPU

La probabilidad de que se esté ejecutando algún proceso es $1 - p^n$, donde n es la cantidad de procesos.

Multiple Fixed Partitions (IBM OS/MFT)

Multitasking with a Fixed number of Tasks (MFT):

- ➢ Cada partición era monoprogramada (cuando el SO se instalaba o se redefinían por el operador)
- ➢ La multiprogramación ocurría entre las particiones
- ➢ Los límites de las particiones no eran móviles
- ➢ Las particiones eran de tamaño fijo
- ➢ La fragmentación era un problema
- ➢ No existía el concepto de memoria compartida

Multiple Variable Partitions (IBM OS/MVT)

Multitasking with a Variable number of Tasks (MVT):

- ➢ Cada tarea podía tener cualquier tamaño
- ➢ La limitante era la memoria de la máquina
- ➢ Una única cola de procesos listo para correr
- ➢ La tarea podía ser movida
- ➢ Se elimina la fragmentación interna
- ➢ Introduce la fragmentación externa

Time Sharing

Compartir de forma concurrente un recurso computacional entre muchos usuarios por medio de tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del SO permitiendo a este acotar el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado.

Workload

Carga de trabajo de un proceso corriendo en el sistema. Cuanto mejor es el cálculo, mejor es la política de planificación.
Los supuestos sobre los procesos o *jobs* que se encuentran en ejecución son:

1. Cada uno se ejecuta la misma cantidad de tiempo
2. Todos llegan al mismo tiempo para ser ejecutados

3. Una vez que empieza uno, sigue hasta completarse
4. Todos usan únicamente CPU
5. El tiempo de ejecución (run-time) de cada uno es conocido

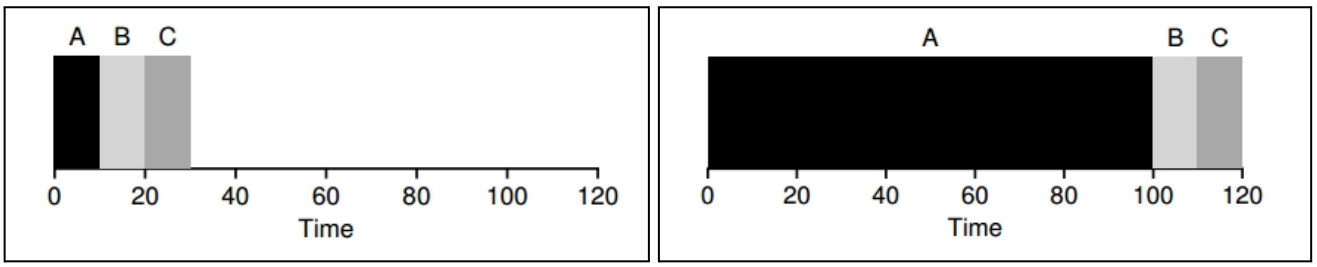
Métricas de Planificación

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Políticas para Sistemas Mono-Procesador

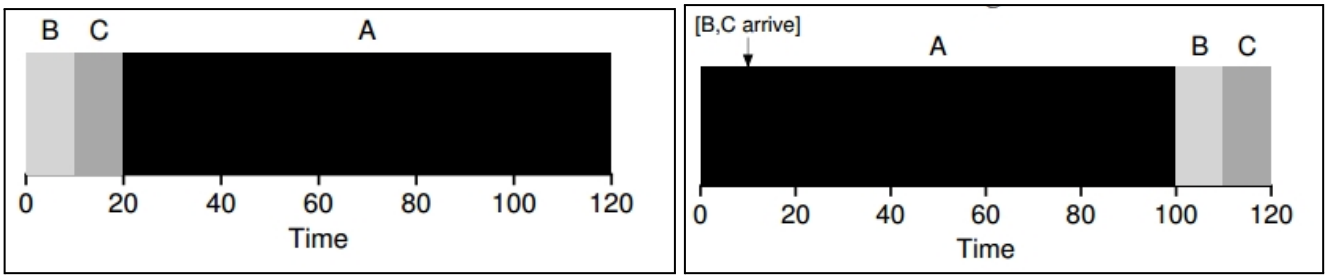
FIFO - First In First Out

Es simple, fácil de implementar y funciona para las suposiciones iniciales.
Falla al relajar la suposición 1: se produce un efecto convoy.



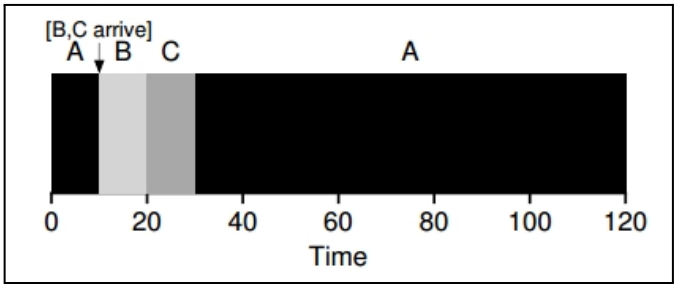
SJF - Shortest Job First

Se ordenan los procesos por tiempo de duración de menor a mayor.
Falla al relajar la suposición 2: el proceso que llega primero es el más largo.



STCF - Shortest Time to Completion

Al relajar la suposición 3: el planificador o sheduler puede adelantarse y determinar qué proceso debe ser ejecutado. Es una política preemptive.



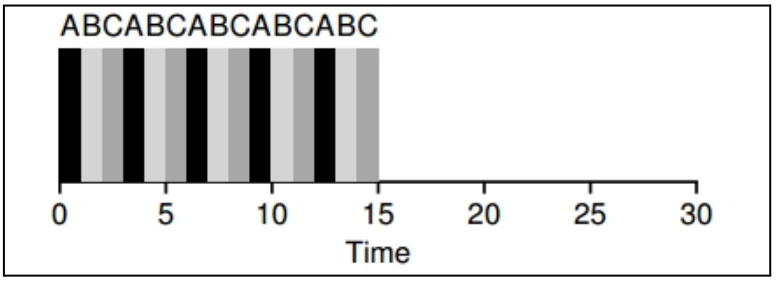
Una nueva métrica: Tiempo de Respuesta

De la mano de time sharing: los usuarios pretenden una interacción con rapidez.

Nace el response time como métrica: $T_{response} = T_{firstrun} - T_{arrival}$

Round Robin (RR)

1. se ejecuta un proceso por un período determinado de tiempo (slice)
2. transcurrido el período se pasa a otro proceso
3. repito cambiando de proceso en la cola de ejecución



Es importante la elección de un buen time slice: tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

Multi-Level Feedback Queue (MLFQ)

Técnica de planificación que intenta atacar principalmente 2 problemas:

1. Optimizar el turnaround time que se realiza mediante la ejecución de la tarea más corta primero (el SO nunca sabe cuánto va a tardar una tarea)
2. Que el planificador haga sentir al SO con un tiempo de respuesta interactivo para los usuarios y así minimizar el response time (los algoritmos como RR reducen el response time pero tienen un terrible turnaround time)

Reglas Básicas

Conjunto de distintas colas, cada una con un nivel de prioridad. En un determinado tiempo, una tarea que está lista para ser corrida está en una única cola. Se usan las prioridades para decidir cuál tarea debería correr en un determinado tiempo:

- **Regla 1**: if (PA >= PB) then exec(A)
- **Regla 2**: if (PA == PB) then RR(A,B)

El planificador establece las prioridades basándose en el comportamiento observado:

➔ Si no utiliza la CPU, mantiene su prioridad alta

➔ Si utiliza intensivamente por largos períodos de tiempo el CPU, reducirá su prioridad

El ajuste se realiza según:

- **Regla 3**: cuando entra en el sistema se pone con la más alta prioridad
- **Regla 4a**: si usa un time slice mientras se está ejecutando su prioridad se reduce en una unidad
- **Regla 4b**: si renuncia al uso de la CPU antes de un time slice completo se queda en el mismo nivel de prioridad

Problema

1. Starvation: si hay demasiadas tareas interactivas se van a combinar para consumir todo el tiempo del CPU y las de larga duración nunca se van a ejecutar.

2. Se podrían reescribir los programas para obtener más tiempo de CPU.

Segundo Approach

- **Regla 5**: después de cierto período de tiempo *S* se mueven a la cola con más prioridad.

Así,

- se garantiza que los procesos no van a starve -> al ubicarse en la cola tope con las otras de alta prioridad se van a ejecutar utilizando RR y en algún momento recibirá atención,
- si un proceso que consume CPU se transforma en interactivo el planificador lo tratará como tal una vez que haya recibido el boost de prioridad.

El valor del tiempo *S* deberá ser determinado correctamente: si es demasiado alto los procesos que requieren mucha ejecución van a caer en starvation; si es muy pequeño las tareas interactivas no van a poder compartir adecuadamente la CPU.

Gaming

Para prevenir que ventajeen al planificador: se lleva una mejor contabilidad del tiempo de uso de la CPU en todos los niveles de la MLFQ. Reescribiendo:

- **Regla 4**: una vez que una tarea usa su asignación de tiempo en un nivel dado (sin importar cuántas veces renunció al uso de la CPU) su prioridad se reduce.

Planificación: Proportional Share

Se intenta garantizar que cada tarea tenga cierto porcentaje de tiempo de CPU. Cada tanto se realiza un sorteo para determinar qué proceso tiene que ejecutarse a continuación tal que los que deban ejecutarse con más frecuencia tienen que tener más probabilidades de *ganar la lotería*.

Concepto

Los boletos son utilizados para representar cuánto se comparte de un determinado recurso para un determinado proceso. El porcentaje de los boletos que un proceso tiene es el porcentaje de cuánto va a compartir el recurso en cuestión.

Mecanismo

★ Ticket Currency: existen distintos tipos de moneda y las tareas pueden tener los tickets comprados con distintos valores, el sistema automáticamente los transforma en un tipo global de moneda

★ Transferencia de Boletos: permite que un proceso temporalmente transfiera sus boletos a otro proceso, es útil cuando se está utilizando la arquitectura cliente/servidor

★ Inflación: un proceso puede aumentar o disminuir la cantidad de boletos que posee de forma temporal, no puede realizarse en un sistema en el cual las tareas compiten entre ellas ya que una tarea muy avara podría captar todos

los boletos; puede ser utilizado en un ambiente en el cual los procesos confían entre ellos

Implementación

1. Sortear un boleto
2. Buscar en la lista de procesos quién tiene el número ganador

Planificación Avanzada: Planificación Multiprocesador

Procesadores multi-núcleo: múltiples núcleos de CPU están empaquetados en un único chip.

Arquitectura Multiprocesador

En un sistema con único CPU la **caché** suele ayudar al procesador a correr los programas más rápidamente pues es una memoria pequeña y rápida donde se almacenan copias de datos de memoria principal que son comúnmente utilizados.
La **memoria principal** mantiene todos los datos del sistema pero el acceso a los mismos es lento.
La caché se basa en la noción de localidad:

- **Temporal**: cuando cierta cantidad de datos son accedidos, es muy probable que sean accedidos otra vez en un futuro cercano.
- **Espacial**: un programa que accede a una dirección es muy probable que necesite volver a acceder cerca de la misma.

Cuando múltiples procesadores tienen que compartir una única memoria principal, el hardware monitorea los accesos a memoria asegurando que la vista de una única memoria compartida sea preservada. Una forma de hacerlo es mediante 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 ve que se actualizó un dato que está mantenido en su propio caché esta se va a dar cuenta del cambio y va a invalidar su copia o actualizarlo.

Afinidad de Caché

Debería considerarse la afinidad de caché durante la toma de decisiones de planificación de ejecución de procesos manteniendo a un proceso en un determinado CPU si es posible.

Single Queue Multiprocessor Scheduling (SQMS)

Se ponen todos los trabajos que tienen que ser planificados en una única cola.
Limitaciones:

- ➢ No es escalable.
- ➢ Para que trabaje correctamente en una arquitectura multiprocesador, los desarrolladores tienen que insertar algún tipo de bloqueo en su código fuente que asegure que cuando SQMS accede a una única cola, un resultado

correcto se ha obtenido. El bloqueo reduce la performance (proporcional a la cantidad de CPUs en la arquitectura).
- ➢ No implementa la idea de afinidad de caché.

Multi-Queue Planification

Cada cola sigue una determinada disciplina de planificación -> cuando una tarea entra en el sistema ésta se coloca exactamente en una única cola de planificación de acuerdo con alguna heurística y es planificada de forma independiente.

- ➔ Es escalable: las colas crecen con las CPU.
- ➔ Provee afinidad de caché: las tareas intentan mantenerse en la CPU en la que fueron planificadas.
- ➢ *Load Imbalance*: se da cuando una CPU queda ociosa frente a las demás que están sobrecargadas. Podría resolverse con migración o migration (se migra la tarea a otra CPU consiguiendo un balance de carga).

IRL

Linux: Completelly Fair Scheduler (CFS)

Planificador que implementa *fair-share scheduling* de una forma altamente eficiente y de forma escalable, gastando muy poco tiempo tomando decisiones de planificación tanto por su diseño como por el uso inteligente de estructuras de datos para esa tarea.

Modo de Operación Básico

Su objetivo es dividir de forma justa la CPU entre todos los procesos que están compitiendo por ella utilizando **virtual runtime** (runtime normalizado por el número de procesos runnable).
A medida que un proceso se ejecuta acumula **vruntime**. Cuando una decisión de planificación ocurre, CFS seleccionará el proceso con menos vruntime para que sea el próximo ejecutado. Los parámetros de control que utiliza para decidir cuándo parar de ejecutar un proceso son:

1. **sched_latency**: determina por cuánto tiempo un proceso tiene que ejecutarse antes de considerar su switcheo (como un time-slice pero dinámico).
2. **min_granularity**: determina el tiempo mínimo que un proceso debe ejecutarse (CFS nunca sede el time-slice de un proceso por debajo de ese número y se asegura que no haya overhead por context switch).

Weighting (niceness)

CFS controla las prioridades de los procesos tal que los usuarios y administradores puedan asignar más CPU a un determinado proceso mediante un mecanismo cálcico de Unix llamado nivel de proceso *nice*: valor que va de -20 a +19 con un valor por defecto de 0 (los valores positivos implican prioridad más baja y valores negativos implican prioridad más alta). La tabla conserva las

proporciones de ratio de la CPU cuando la diferencia en valores de nice es constante.
Estos pesos permiten calcular el time slice para cada proceso teniendo en cuenta las distintas prioridades para cada uno:

$$time\_slice_k = \frac{weight_k}{\sum_{n=0}^{n-1} weight_i} * sched\_latency$$

Con esto se generaliza el cálculo del vruntime:

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

Utiliza Árboles Rojo-Negro

Para la búsqueda del próximo job a ser ejecutado: el árbol rojo-negro escala en O(logn).

El Algoritmo

Cuando el scheduler es invocado para correr un nuevo proceso:

1. El nodo más a la izquierda del árbol de planificación es elegido por tener el tiempo de ejecución más bajo y es enviado a ejecutarse.
2. Si el proceso simplemente completa su ejecución, este es eliminado del sistema y de árbol de planificación.
3. Si el proceso alcanza su máximo tiempo de ejecución o se detiene la ejecución del mismo (ya sea voluntariamente o vía una interrupción) este es reinsertado en el árbol de planificación basado en su nuevo tiempo de ejecución (vruntime).
4. El nuevo nodo que se encuentre más a la izquierda del árbol será ahora seleccionado y se repite la iteración.

La Memoria

El Espacio de Direcciones o Address Space

Proceso que contiene todo el estado de la memoria de un programa en ejecución:

- El **código fuente** del programa vive en lo alto del espacio de direcciones. Se puede poner allí pues es estático y no necesitará más espacio mientras que el programa se ejecute.
- Mientras se esté ejecutando, el **heap** y el **stack** pueden crecer o achicarse y se colocan en los extremos del espacio de direcciones enfrentados entre sí permitiendo su crecimiento en direcciones opuestas (el heap empieza después del código fuente y crece hacia abajo y el stack empieza al final del espacio de direcciones y crece hacia arriba).

Abstracción fundamental sobre la memoria de una computadora que el SO le provee al programa en ejecución (aka: virtualización de memoria).

Metas

➔ Transparencia: el programa debe comportarse como si él estuviera alojado en su propia área de memoria física privada.
➔ Eficiencia: en términos de tiempo y espacio para no hacer que los programas corran más lentos ni usar demasiada memoria para las estructuras de virtualización.
➔ Protección: mediante el aislamiento entre procesos (cada uno tiene que ejecutarse en su propio caparazón aislado y seguro de los avatares de otros con fallas o incluso maliciosos).

El API de Memoria

Tipos de Memoria

★ Memoria de Stack: su reserva y liberación es manejada por el compilador en nombre del programador.
★ Memoria de Heap: es obtenida y liberada explícitamente por el programador.

void *malloc(size_t size)

❖ Devuelve un puntero a un bloque de memoria de por lo menos *size* bytes alineado a cualquier tipo de datos que pueda contener el bloque
  ➢ 32 bits -> devuelve valores de direcciones múltiplos de 8
  ➢ 64 bits -> devuelve valores de direcciones múltiplos de 16
❖ Devuelve NULL si algo salió mal y setea errno
❖ No inicializa el bloque de memoria

void free(void *ptr)

❖ *ptr* debe ser un bloque devuelto por malloc(), calloc() o realloc()
❖ No avisa si algo salió mal

Teorema de Méndez

La memoria se crea y se destruye, nunca se transforma.

Implementación

Se utilizan las syscalls brk() y sbrk()

brk (break)

Variable que mantiene el Kernel para cada proceso que apunta al tope del heap (donde termina).

int brk(void *addr)

Sustituye la dirección del brk con el valor especificado en *addr* (nueva dirección de fin del heap).

int sbrk(int size)

Hace lo mismo que brk() pero el parámetro es la cantidad de bytes. Llama a brk().

Address Translation

Mecanismo proporcionado por el hardware que permite -a partir de una dirección virtual- obtener la dirección física correspondiente. El SO está en el medio determinando si la traducción se realizó correctamente gerenciando el manejo de memoria pues:

- mantiene un registro de qué parte está libre
- mantiene un registro de qué parte está ocupada
- mantiene el control de la forma en la cual la memoria está siendo utilizada

Provee:

- **Process Isolation**: used by applications to construct safe execution sandboxes for third party extensions.
- **Interprocess Communication**: sharing a common memory region.
- **Shared Code Segments**: different programs can share common libraries (instances of the same program) reducing their memory footprint & making the processor cache more efficient.
- **Program Initialization**: start a program running before all of its code is loaded into memory from disk.
- **Efficient Dynamic Memory Allocation**: trapping the Kernel to allocate memory as a process grows its heap or as a thread grows its stack only as needed.
- **Cache Management**: the OS arranges how programs are positioned in physical memory to improve cache efficiency through page coloring.
- **Program Debugging**: to prevent a buggy program from overwriting its own code region by catching pointer errors earlier & also installing data breakpoints.
- **Efficient I/O**: enables data to be safely transferred between user-mode applications an I/O devices.

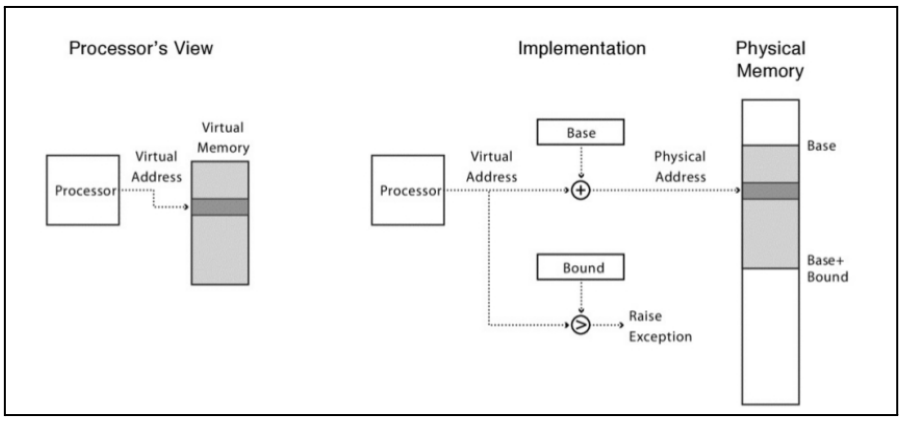
- **Memory Mapped Files**: mapping files into the address space so that the contents can be directly referenced with program instructions.
- **Virtual Memory**: providing applications the abstraction of more memory than is physically present on a given computer.
- **Checkpointing & Restart**: periodically checkpointing a long-running program so that if it crushes it can be restarted from the saved state.
- **Persistent Data Structures**: providing the abstraction of persistent region of memory where changes to the data structures survive program & system crashes.
- **Process Migration**: an executing program can be transparently moved from one server to another.
- **Information Flow Control**: to verify that a program is not sending private data to a third party.
- **Distributed Shared Memory**: turning a network of servers into a large-scale shared-memory parallel computer.

Dynamic Realocation o Memoria Segmentada [DAH]

Base & Bound

Primera implementación de hardware-based address translation. Sólo necesita dos registros dentro de cada CPU: registro base y registro límite o segmento (ambas contenidas en la MMU). Permite que el address space pueda ser ubicado en cualquier lugar de la memoria física mientras que el SO se asegura que el proceso sólo puede acceder a su address space (protección de memoria). La ubicación sucede en run-time y debido a que se puede mover el address space (incluso una vez que el proceso comenzó a ejecutarse) la técnica es referida como dynamic realocation.
Cada programa es escrito y compilado como si fuera cargado en la dirección física 0 y cuando se inicia la ejecución el SO decide en qué lugar va a cargarlo seteando el registro base. Cuando el proceso genera una referencia, el procesador la traducirá como: $physical\,address = virtual\,address + base$.
Si un proceso genera una dirección virtual que es mayor que los límites o una dirección que es negativa, la CPU genera una excepción y el proceso finaliza.
El bound register puede ser definido:

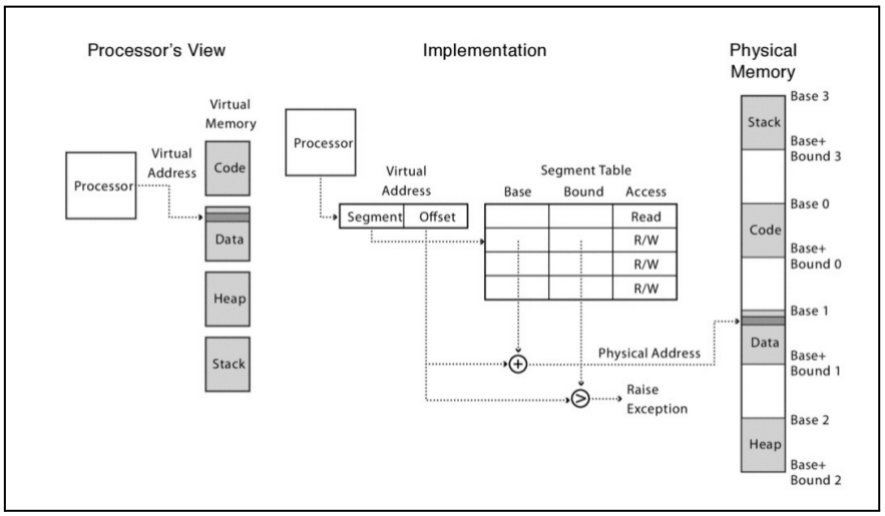
1. mantiene el tamaño del address space -> el hardware le suma primero el registro base.
2. almacena la dirección física del fin del espacio de direcciones.



Problema

Se tiene un sólo registro base y un sólo segmento.

Tabla de Segmentos



Arreglo de pares de (registro base, segmento) por cada proceso, cada entrada controla una porción virtual del address space. La memoria física de cada segmento es almacenada continuamente pero distintos segmentos pueden estar ubicados en distintas partes de la memoria física.
Una memoria virtual tiene dos componentes:

- Número de segmento: índice de la tabla para ubicar el inicio del segmento en la memoria física.
- Offset de segmento: sumado al registro base y comparado contra el registro bound para que el proceso no acceda a regiones indebidas de memoria.

Segmentation Fault

Error que se daba cuando -en las máquinas que se implementaban segmentación- se quería direccionar una posición fuera del espacio direccionable.

Problema

Fragmentación externa.

Notas sobre Segmentación

1. El stack crece backward -> el hardware necesita saberlo en un bit de información.
2. Compartir segmentos -> se utiliza el **protection bit** que indica si se puede ejecutar, escribir o leer por varios procesos a la vez.
3. MMU -> realiza todas las operaciones de segmentación.
4. Segmentación de grano fino vs de grano grueso -> la primera consiste en tener muchos segmentos pequeños y la segunda en tener pocos segmentos grandes.

Memoria Paginada

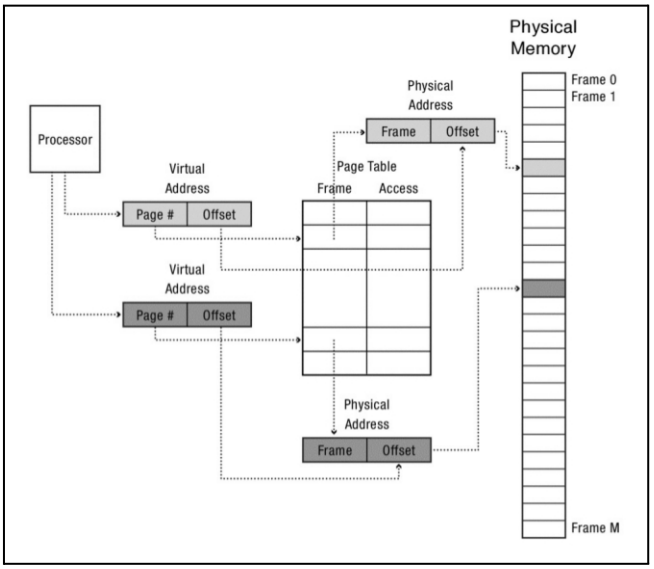
La memoria es reservada en pedazos de tamaño fijo (potencia de 2): **page frames**. Se tiene una tabla de páginas por cada proceso cuyas entradas contienen punteros a las page frames.

Address Translation con Page Table

La memoria virtual tiene dos componentes:

1. Número de página virtual: índice en la page table para obtener el page frame en la memoria física.
2. Offset dentro de esa página: sumado a la page table compone la physical frame page.

Si bien el programa cree que su memoria es lineal, está desparramada por toda la memoria física. El procesador ejecuta instrucciones usando direcciones virtuales lineales.



Problema

Saber qué espacio de la memoria está vacío es complicado.

Translation multilevel

La implementación mediante árbol o hash soporta:

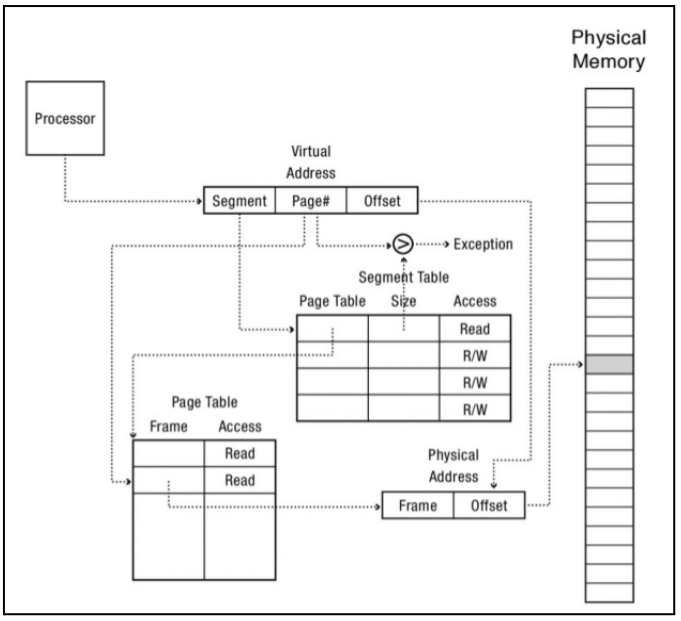
- Protección de memoria de grano fino
- Memoria compartida
- Ubicación de memoria flexible
- Reserva eficiente de memoria
- Sistema de búsqueda de espacio de direcciones eficiente

Paged Segmentation

La memoria está segmentada y cada entrada en la tabla de segmentos apunta a una tabla de páginas que a su vez apunta a la memoria correspondiente a ese segmento. La tabla de segmentos tiene una entrada llamada límite o tamaño que describe la longitud de la página de tablas.

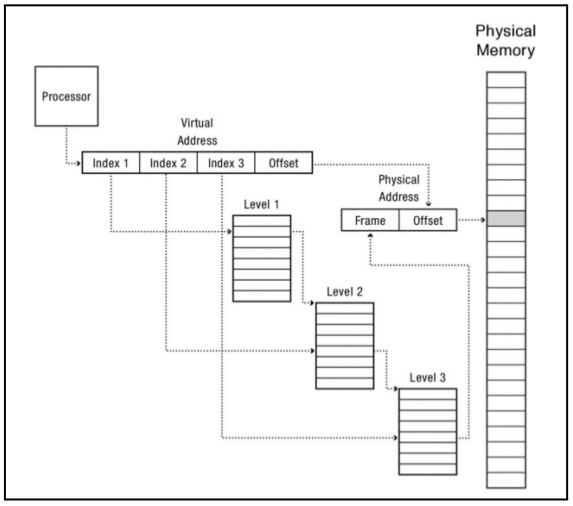
La memoria virtual tiene 3 componentes:

1. Número de segmento: índice para la segment table que aloja la page table para ese segmento.
2. Número de página virtual dentro de ese segmento: índice de la page table que contiene una page frame en la memoria física.
3. Offset dentro de la página: sumado al physical page frame que pertenece a la page table compone la physical address.

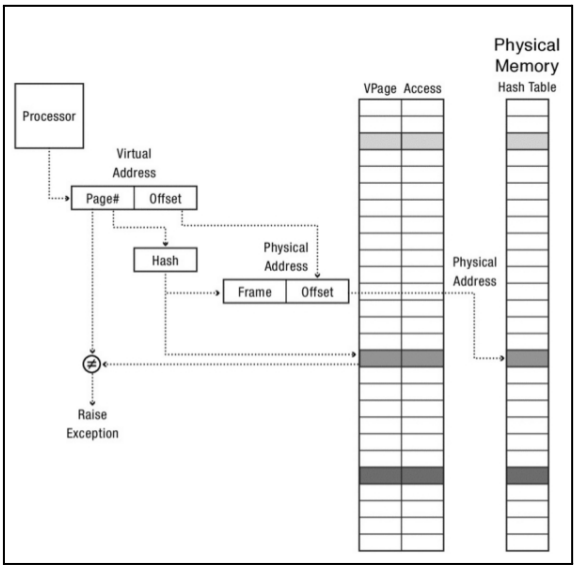


Address Translation con Tres Niveles de Page Tables

Implementa múltiples niveles de page tables.



Address Translation con Tabla de Hash por Software



Multilevel page Segmentation

Arquitectura x86

Cada proceso posee una Global Descriptor Table (GDT) almacenada en memoria equivalente a la segment table. Cada entrada a esta tabla apunta a una tabla de páginas (multinivel) para ese segmento. Para inicializar el proceso de address translation el sistema operativo setea la GDT e inicializa el registro GDTR que contiene la dirección y la longitud de la GDT. Por cada 32 bit la virtual address posee un segmento a una tabla de 2 niveles:

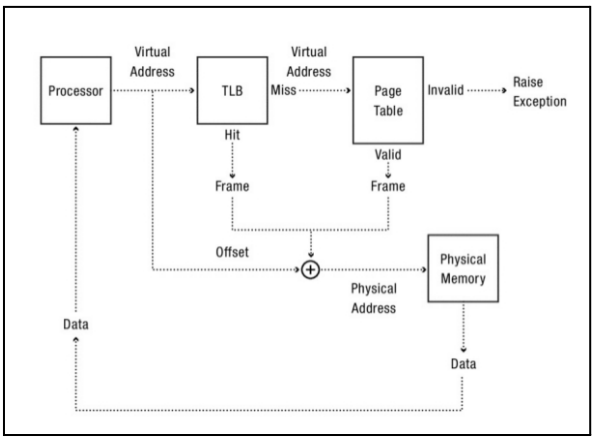
1. Los primeros 10 bit de la virtual address son el índice de la paged table de primer nivel: page directory
2. Los otros 10 bit son el índice de una page table de segundo nivel

Los 12 bit restantes son el offset dentro de la página.

Hacia una eficiente Address Translation

Para mejorar el rendimiento de la traducción de las direcciones se utilizará un caché que consiste en una copia de ciertos datos que pueden ser accedidos más de una vez más rápidamente.

Lookaside buffer



Pequeña tabla a nivel hardware (implementado en memoria estática on-chip localizada muy cerca del procesador) que contiene los resultados de las recientes traducciones de memorias realizadas. Cada entrada de la tabla mapea una virtual page a una physical page.
Se chequean todas las entradas de la TLB contra la virtual page:

- ➔ **TLB hit**: existe matcheo -> el procesador lo utiliza para formar la physical address ahorrándose los pasos de la traducción
- ➔ **TLB miss**: no existe matcheo

Consistencia de la TLB

El SO tiene que asegurarse que cada programa ve sólo su memoria:

- **Context Switch**: cada vez que ocurre se descarta el contenido de la TLB (flush de TLB) pues las direcciones virtuales del viejo proceso no son válidas para el nuevo.
- **Reducción de Permiso**: mantener la TLB consistente con la page table es responsabilidad del SO.
- **TLB shutdown**: cada vez que una entrada en la page table es modificada, la correspondiente entrada en todas las TLB de los procesadores tiene que ser descartada antes que los cambios tomen efecto. Para eliminar una entrada en todos los procesadores del sistema el SO manda una interrupción a cada procesador y pide que esa entrada de la TLB sea eliminada.

La TLB

Ubicada en cercanías del núcleo que realiza el procesamiento y está diseñada para ser muy rápida.
Su mecanismo de funcionamiento es: se extrae la virtual page number (VPN) de la virtual address y se chequea si la TLB tiene esa traducción para esa VPN

1. TLB HIT: existe la traducción
    a. se extrae el Page Frame Number (PFN) de la entrada en la TLB
    b. se concatena con el offset de la virtual address original
    c. se conforma la physical address para acceder a la memoria
2. TLB MISS: no existe la traducción
    a. el hardware accede la page table para encontrar la traducción

b. si la referencia a la memoria virtual generada por el proceso es válida y accesible se actualiza la TLB con la traducción hecha por el hardware
c. el hardware vuelve a buscar la instrucción en la TLB y la referencia a la memoria es procesada rápidamente

Concurrencia

La Abstracción

Un **thread** es una secuencia de ejecución atómica que representa una tarea planificable de ejecución.

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

Thread vs Procesos

➔ Proceso: programa en ejecución con derechos restringidos
➔ Thread: secuencia independiente de instrucciones ejecutándose dentro de un programa que se caracteriza por
  - id
  - conjunto de valores de registros
  - stack propio
  - política y prioridad de ejecución
  - propio errno
  - datos específicos

One thread per process

Proceso con una única secuencia de instrucciones ejecutándose de inicio a fin.

Many thread per process

Un programa es visto como threads ejecutándose dentro de un proceso con derechos restringidos.

Many single-threaded processes

Limitación de algunos SO que permitían varios procesos, cada uno con un único thread, resultando varios threads ejecutándose en kernel mode.

Many kernel threads

El kernel puede ejecutar varios threads en kernel mode.

Thread Scheduler

El cambio entre threads es transparente: el programador no se ocupa de la suspensión de threads.
Cada thread uno corre en un procesador virtual exclusivo con una velocidad variable e impredecible: desde el punto de vista del thread, cada instrucción se ejecuta inmediatamente una detrás de otra.

Hay dos formas de que los threads se relacionen entre sí:

1. **Multi-threading Cooperativo**: no hay interrupción a menos que se solicite
2. **Multi-threading Preemptivo**: un thread en estado de running puede ser movido en cualquier momento

El API de Threads

Creación

*int pthread_create(pthread_t * thread, const pthread_att_t * att,*
*void * (start_routine) (void *), void * arg)*

- *thread*: puntero a estructura *pthread_t* para interactuar con el thread
- *attr*: especifica los atributos que el thread debería tener (ej: tamaño del stack, la prioridad de scheduling del thread)
- *start_routine*: puntero a una función
- *arg*: puntero a los argumentos de la función

Devuelve 0 si se ha creado el thread con éxito, en otro caso hubo error.

Terminación

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

- *thread*: thread por el que hay que esperar
- *value_ptr*: puntero al valor esperado de retorno

Estructura y Ciclo de Vida de un Thread

El SO guarda y carga el estado de cada thread. Existen dos estados:

1. Per thread
2. Compartido entre varios threads

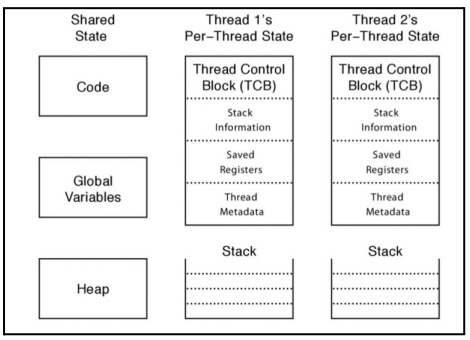
TCB: El Estado Per-thread y Threads Control Block

Estructura que representa el estado de un thread (se crea una entrada por cada uno) y almacena el estado per-thread de cada thread: el estado del cómputo que debe ser realizado por el thread.
El SO almacena el estado actual del bloque de ejecución:

- El puntero al stack del thread
- Una copia de sus registros en el procesador

Metadata referente al thread que es utilizada para su administración

Por cada thread se guarda: ID, Prioridad de Shceduling y Status.

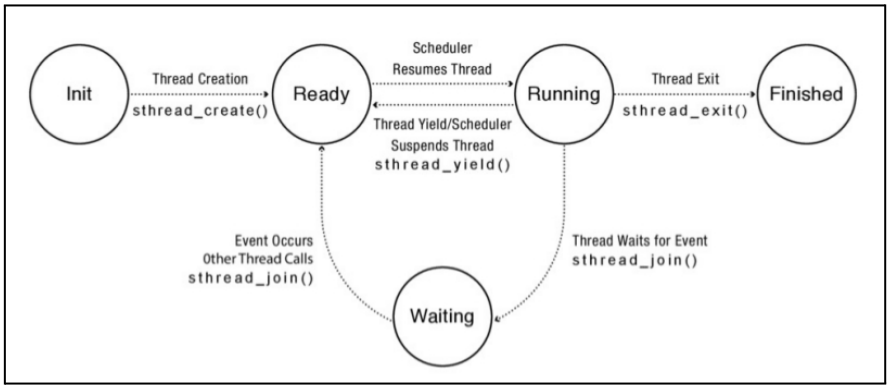


Shared State o Estado Compartido

Contrariamente al per-thread state, se guarda cierta información que es compartida por varios threads: código, variables globales y del heap.

Estados de un Thread

- ★ **Init**: mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras.
- ★ **Ready**: listo para ser ejecutado. Se lo coloca en la *ready list* y los valores de los registros están en la TCB.
- ★ **Running**: siendo ejecutado en este mismo instante por el procesador, donde almacena los valores de los registros. Puede pasar a Ready de dos formas:
  - El scheduler lo desaloja guardando los valores de los registros y cambiando el thread que se está ejecutando por el próximo de la lista
  - Voluntariamente solicita abandonar la ejecución (ejemplo: con *thread_yield*)
- ★ **Waiting**: esperando que algún evento suceda. Se almacenan en la *waiting list* y el scheduler pasa el thread del estado *Waiting* a *Running* (mueve la TCB desde el *waiting list* a la *ready list*).
- ★ **Finished**: nunca más podrá volver a ser ejecutado. Se agrega a la *finished list* en la que se encuentran las TCB de los threads que han terminado.



Threads y Linux

Diferencias Proceso/Thread

Threads

- ❖ Comparten memoria
- ❖ Comparten los descriptores de archivos
- ❖ Comparten el contexto del filsystem
- ❖ Comparten el manejo de señales

Procesos

- ❖ No comparten memoria
- ❖ No comparten los descriptores de archivos
- ❖ No comparten el contexto del filsystem
- ❖ No comparten el manejo de señales

Sincronización

Race Conditions

Se da cuando el resultado de un programa depende de cómo se intercalaron las operaciones de los threads que se ejecutan dentro de ese proceso.

Operaciones Atómicas

No pueden dividirse en otras y se garantiza la ejecución de la misma sin tener que intercalar ejecución.

Locks

Variable que permite la sincronización mediante la **exclusión mutua**: cuando un thread tiene el candado o lock ningún otro puede tenerlo. Todo lo que se ejecuta en la región de código en la cual un thread tiene un lock, garantiza la atomicidad de las operaciones.
Tiene dos estados:

- Busy
- Free

Propiedades

Asegura:

- ➔ Exclusión Mutua: como mucho un solo thread posee el lock a la vez.
- ➔ Progress: si nadie posee el lock alguno debe poder obtenerlo.
- ➔ Bounded Waiting: si T quiere acceder al lock y existen varios en la misma situación, los demás tienen una cantidad finita de posibles accesos antes que T lo haga.

Sección Crítica

Sección del código fuente encerrada dentro de un lock que se necesita ejecutar en forma atómica.

Condition Variables: Esperando por un cambio

Permiten a un thread esperar a otro para tomar una acción y compartir algún estado que está protegido por un lock.
Tiene tres métodos:

1. *wait(Lock *lock)*:
    a. suelta el mutex haciendo *unlock*
    b. suspende la ejecución del thread que lo llama poniéndolo en la lista de espera de la *condition variable*
    c. se vuelve a hacer lock del mutex antes de volver del *wait*

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

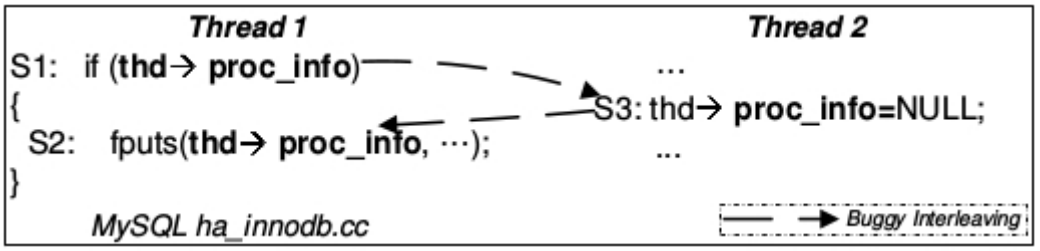
Errores comunes de concurrencia

Existen dos tipos:

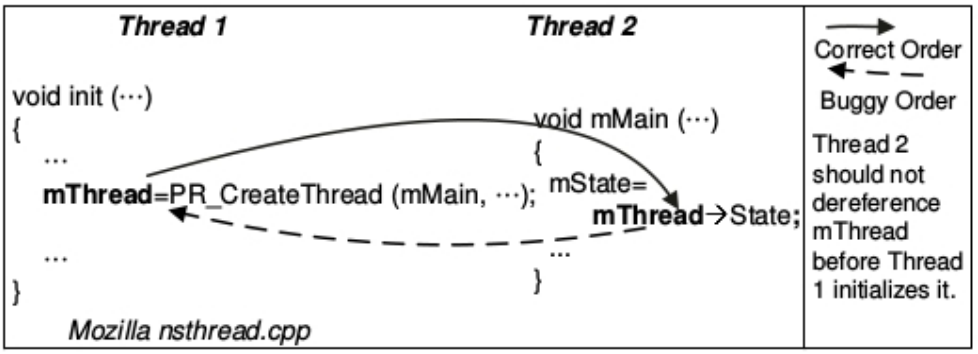
1. Non-deadlock Bugs
2. Deadlock Bugs

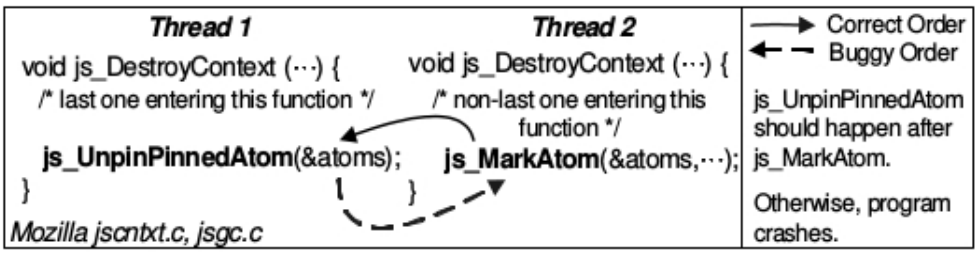
Non-Deadlock Bugs

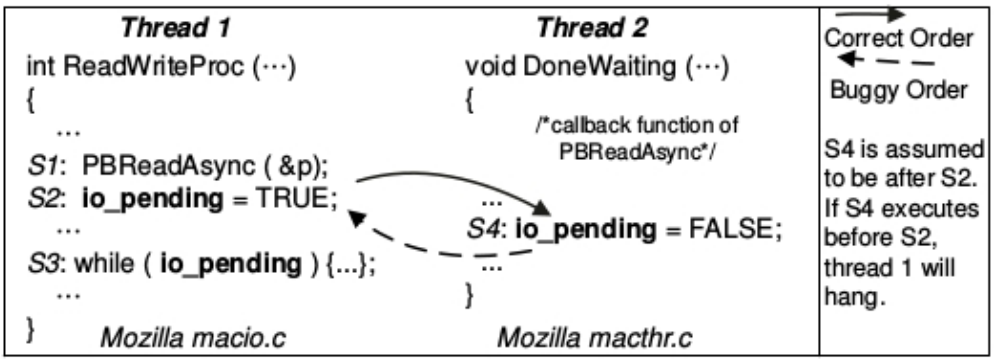
➢ **Atomicity violation**: "el deseo de la serialización entre múltiples accesos a memoria es violado"



➢ **Order Violation**: "El orden deseado entre accesos a memoria se ha cambiado"







DeadLock Bugs

Entre dos o más threads uno obtiene el lock y por algún motivo nunca libera el mismo haciendo que sus compañeros se bloqueen.

Condiciones Necesarias para que se dé un DeadLock

- ➔ **Exclusión mutua**: los threads reclaman control exclusivo sobre un recurso compartido que necesitan (para comer los fideos son necesarios dos palitos).
- ➔ **Hold-and-Wait**: un thread mantiene un recurso reservado para sí mismo mientras espera que se dé alguna condición (si un filósofo necesita esperar por un palito, mantendrá el que tenga en la otra mano).
- ➔ **No preemption**: los recursos adquiridos no pueden ser desalojados (preempted) por la fuerza (una vez que un filósofo consigue un palito no lo liberará hasta que haya terminado de comer).
- ➔ **Circular wait**: existe un conjunto de threads que de forma circular cada uno reserva uno o más recursos que son requeridos por el siguiente en la cadena (cada filósofo esperará que el vecino de la izquierda libere el palito).