



# **Sistemas Operativos**

## **Scheduler II**

---

# Caso de estudio: Linux

## Conceptos clave:

- Completely Fair Scheduler
- vruntime
- nice() y prioridades
- Latencia objetivo
- Scheduling de grupos y cgroups

# Linux



El planificador de tareas de Linux desde su primer versión en 1991 pasando por todas **hasta el kernel versión 2.4 fue sencillo** y casi pedestre en lo que respecta a su diseño [LOV], cap. 4.

**Round-Robin:** El método básico de planificación era round-robin, que asigna a cada proceso un intervalo de tiempo fijo, llamado quantum de tiempo, durante el cual el proceso puede ejecutar. Una vez que el quantum de tiempo de un proceso se agota, el scheduler pone al proceso en espera y pasa al siguiente proceso en la cola.

**Multinivel de Feedback Queue:** Linux utilizaba un enfoque de múltiples colas para manejar procesos con diferentes prioridades. Los procesos se organizaban en diferentes colas basadas en sus prioridades y comportamiento (por ejemplo, si un proceso utilizaba mucho el CPU o si cedía frecuentemente el CPU). Un proceso podía moverse entre colas si su comportamiento cambiaba, lo que permitía al sistema adaptarse dinámicamente y dar más tiempo de CPU a los procesos que parecían necesitarlo más urgentemente.

# Linux



## Linux 2.4 y versiones anteriores - $O(n)$

Las primeras versiones de Linux utilizaban un scheduler básico basado en "**round-robin**", que simplemente asignaba el CPU a los procesos en un ciclo fijo, permitiendo que cada proceso se ejecutara por un **timeslice** determinado antes de pasar al siguiente. Aunque este enfoque era simple, resultaba ineficiente a medida que los sistemas se volvieron más complejos y con una mayor carga de procesos.

## Linux 2.6 - $O(1)$

Introducido inicialmente en la versión 2.5 y estabilizado en la 2.6, el scheduler  $O(1)$  fue diseñado para ser altamente eficiente, con una complejidad constante  $O(1)$ . Esto significa que el tiempo necesario para tomar decisiones de planificación no dependía de la cantidad de procesos en ejecución, lo que mejoró significativamente la escalabilidad y el rendimiento, especialmente en sistemas con una alta carga de trabajo.

# Linux



El quantum (también conocido como time slice o rodaja de tiempo)

## Linux 2.6.23 - **Completely Fair Scheduler** (CFS)

En 2007, con la versión 2.6.23, Linux incorporó el Completely Fair Scheduler (CFS), diseñado por Ingo Molnar. Este scheduler se basa en la planificación proporcional (**fair scheduling**), donde cada proceso recibe una cantidad de tiempo de CPU acorde a su prioridad. CFS utiliza una estructura de datos tipo **árbol rojo-negro** para gestionar el tiempo de ejecución de los procesos, permitiendo una asignación **más justa y eficiente** del tiempo de CPU, especialmente en sistemas con múltiples núcleos.

## Linux 6.6 - Earliest eligible virtual deadline first scheduling

EEVDF fue descrito por primera vez en el artículo de 1995 titulado "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation" por Ion Stoica y Hussein Abdel-Wahab. Utiliza los conceptos de tiempo virtual, tiempo elegible, solicitudes virtuales y plazos virtuales para determinar la prioridad de programación. Tiene la propiedad de que, cuando un trabajo sigue solicitando servicio, la cantidad de servicio obtenida siempre está dentro del tamaño máximo del quantum al que tiene derecho. El scheduler EEVDF reemplazó al CFS en la versión 6.6 del kernel de Linux.

# Linux



**Prioridades y preemption:** Aunque el núcleo Linux de la serie 2.4 no era completamente preemptivo en su manejo de procesos en espacio de usuario, sí tenía soporte básico para la preemption en el manejo de interrupciones y algunas operaciones críticas del kernel. Esto significa que ciertas tareas críticas podían interrumpir otras menos críticas para mejorar la capacidad de respuesta del sistema. Sin embargo, en el espacio de usuario, un proceso en ejecución no sería necesariamente interrumpido hasta que finalizara su quantum de tiempo, a menos que esperara por I/O u otra interrupción.

# Linux: Principios Básicos del CFS



## Fairness (equidad)

Como funcionaria en teoría:

*Si hay  $n$  procesos cada proceso se ejecutaría en el procesador todo el tiempo a  $1/n$  de la potencia de este.*

Como funciona en la práctica:

*En una ventana de tiempo, el procesador ejecuta cada tarea durante  $1/n$  del tiempo.*

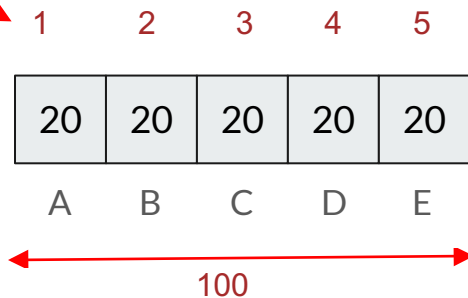
Eg.

Si tengo 5 procesos, en 100 segundos una ejecución “justa” asigna el CPU durante 20 ms a cada proceso.

# Linux: Principios Básicos del CFS

Eg.

Si tengo 5 procesos, en 100 segundos una ejecución “justa” asigna el CPU durante 20 ms a cada proceso.



$$\frac{100}{5} = 20$$



# Linux: Principios Básicos del CFS



**Fairness (equidad):** El CFS busca ofrecer a cada proceso una porción justa del CPU, basándose en el concepto de equidad. No se asignan cuotas fijas de tiempo (time slices), sino que se intenta que cada proceso reciba una porción del tiempo de CPU (proporcional a su peso de prioridad).

El scheduler resuelve un problema de optimización: reducir la varianza del **vruntime** de todos los procesos. Es decir, trata de mantener el **runtime** de cada proceso igual.

# Linux: Principios Básicos del CFS

Algoritmo de selección:

Cada vez que tenga que elegir un proceso, el scheduler elige siempre el que esté más atrás en el vruntime consumido.


Eso es todo!



# Linux: Principios Básicos del CFS

Algoritmo básico:

Ejemplo:



| Proceso | Runtime |
|---------|---------|
| A       | 32      |
| B       | 31      |
| C       | 26      |
| D       | 35      |
| E       | 37      |

## Ejemplo con sched-sim (no CFS)



```
struct proc* select_next(){
    int next_runtime = INT_MAX;
    struct proc* next = NULL;

    for (int i = 0; i < NUMPROC; i++) {
        if (proc[i].status == RUNNABLE) {
            if (proc[i].runtime <= next_runtime) {
                next_runtime = proc[i].runtime;
                next = &proc[i];
            }
        }
    }

    return next;
}
```

# Linux: Principios Básicos del CFS

¿Cómo modelamos el concepto de prioridad?

La prioridad se corresponde con pesos de los procesos. El proceso A pesa más que los demás.

|    |    |    |    |    |
|----|----|----|----|----|
| 40 | 15 | 15 | 15 | 15 |
| A  | B  | C  | D  | E  |

# Linux: Principios Básicos del CFS

¿Cómo modelamos el concepto de prioridad?

El reloj de los procesos con más prioridad va a ir más lento. Entonces como consumen menos vruntime van a ser seleccionados más frecuentemente por el mismo algoritmo.

En consecuencia, terminan teniendo un share mayor del CPU.



# Linux: Principios Básicos del CFS

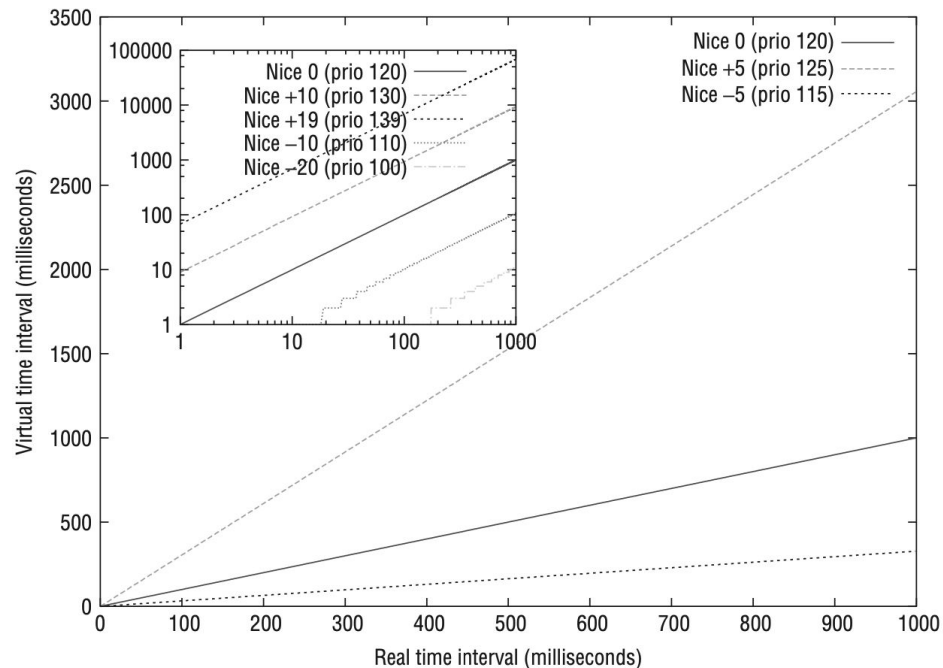


Figure 2-18: Relation between real and virtual time for processes depending on their priority/nice level.

# Linux: Principios Básicos del CFS



Virtual Runtime (vruntime): Cada proceso tiene *asociado un contador llamado vruntime, que representa la cantidad de tiempo que el proceso ha estado ejecutándose en el CPU*. El vruntime se incrementa con cada tick del reloj mientras el proceso está en ejecución.

*Los procesos que consumen vruntime mas lentamente (con respecto al tiempo real) son los que tienen mayor prioridad para ser ejecutados.*



# Linux: vruntime

El cálculo del vruntime es esencial para mantener la equidad en el acceso al procesador entre diferentes procesos. El vruntime *se calcula en función del tiempo que un proceso ha ejecutado en el CPU, ajustado por su prioridad.*

La fórmula general para actualizar el vruntime de un proceso es:

$$\text{vruntime}+ = \frac{\text{delta\_exec}}{\text{weight}}$$

- `delta_exec` es la **cantidad de tiempo** que el proceso ha estado ejecutando desde la última actualización.
- `weight` es el peso asociado a la **prioridad del proceso**.

# Linux: Runqueue



Una **runqueue** (cola de ejecución) es una estructura de datos utilizada por el planificador (scheduler) para gestionar y hacer un seguimiento de todos los procesos (tareas) que están listos para ejecutarse en un núcleo de CPU. Cada núcleo de CPU tiene su propia runqueue, que contiene las tareas que están listas para ejecutar pero que no están ejecutándose en ese momento.

Los procesos ejecutables se colocan en la runqueue, y se extrae de la misma el que tiene menor vruntime. Es esencialmente una **priority-queue**, donde la prioridad es el menor vruntime.

# Linux: Runqueue

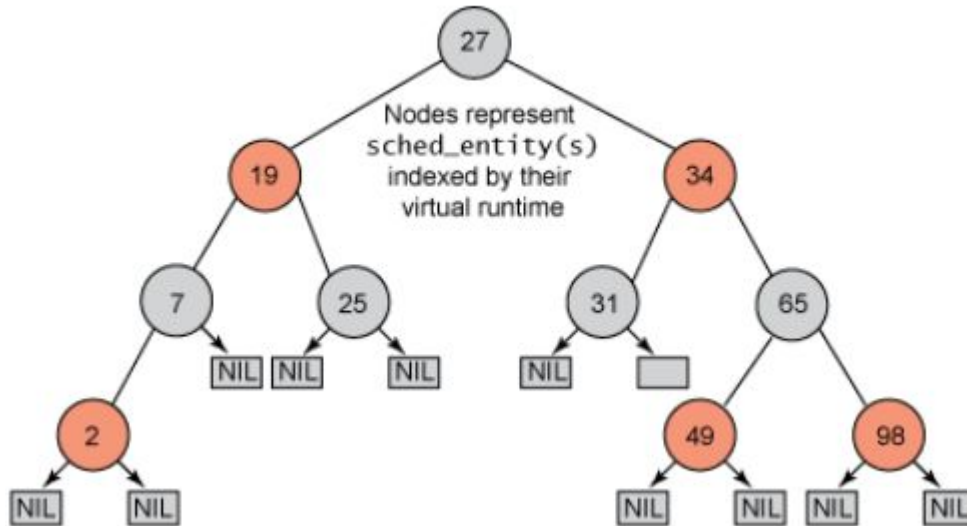


Estamos ignorando un problema importante: ¿Como hacemos para elegir el proceso con menor vruntime de una runqueue, en un tiempo menor que  $O(n)$ ?

Estructuras de datos al rescate!

**Red-Black Tree:** Un red-black tree (árbol rojo-negro) es un tipo de árbol binario de búsqueda auto-balanceado que se utiliza en informática para mantener datos ordenados y permitir inserciones, eliminaciones y búsquedas eficientes. Es una estructura de datos comúnmente utilizada en sistemas operativos, bases de datos y otras aplicaciones que requieren operaciones rápidas de búsqueda y modificación.

# Linux: Arbol Rojo Negro



- Acceso  $O(1)$  al nodo más a la izquierda (tiempo virtual más bajo).
- Inserción  $O(\log n)$ .
- Eliminación  $O(\log n)$ .
- Autoequilibrado.

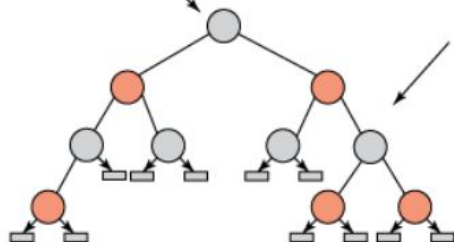
# Linux: Arbol Rojo Negro

```
struct task_struct {  
    volatile long state;  
    void *stack;  
    unsigned int flags;  
    int prio, static_prio normal_prio;  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    ...  
};
```

```
struct cfs_rq {  
    ...  
    struct rb_root tasks_timeline;  
    ...  
};
```

```
struct sched_entity {  
    struct load_weight load;  
    struct rb_node run_node;  
    struct list_head group_node;  
    ...  
};
```

```
struct rb_node {  
    unsigned long rb_parent_color;  
    struct rb_node *rb_right;  
    struct rb_node *rb_left;  
};
```



# Linux: Funcionamiento Detallado



1. **Selección del Proceso:** El scheduler siempre intenta ejecutar el proceso con el **menor vruntime**. Esto asegura que todos los procesos tengan la oportunidad de ejecutarse de manera justa, proporcionando un tiempo de ejecución proporcional a su peso.
2. **Uso de Red-Black Trees:** Para organizar eficientemente los procesos por su vruntime, el CFS utiliza una estructura de datos de **árbol rojo-negro**. Este tipo de árbol permite inserciones, eliminaciones y búsquedas en tiempo logarítmico, lo que es crucial para mantener el rendimiento del scheduler con un gran número de procesos.
3. **Sleeping and Waking Up:** Cuando un proceso se bloquea, por ejemplo, esperando por I/O, se **elimina del árbol rojo-negro**. Cuando se despierta, se vuelve a insertar en el árbol con su vruntime ajustado si es necesario, de manera que no sea penalizado por el tiempo que estuvo bloqueado.

# Linux: Funcionamiento Detallado



1. Cuando un proceso se está ejecutando, su **vruntime** aumentará constantemente, por lo que finalmente se moverá hacia la derecha en el árbol rojo-negro. Debido a que el vruntime aumentará más lentamente para los procesos más importantes, estos también se moverán hacia la derecha más lentamente, lo que aumenta su probabilidad de ser programados en comparación con un proceso menos importante, tal como se requiere.
2. Si un proceso duerme, su vruntime permanecerá sin cambios. Debido a que el `min_vruntime` de la cola (árbol) aumenta mientras tanto, el proceso que estaba dormido se colocará más a la izquierda al despertarse, porque su clave se hizo más pequeña.

# Linux: Funcionamiento Detallado

---

1. **Balance de Carga:** El CFS también se encarga de equilibrar la carga de trabajo entre múltiples CPUs. Esto se logra **migrando procesos** de una CPU a otra si se detecta que una CPU está sobrecargada mientras otra está inactiva o menos cargada.
2. **Soporte para Multi-threading:** CFS maneja los hilos de ejecución (threads) de manera similar a como maneja los procesos. Cada hilo es tratado como un proceso separado con su propio vruntime. Ver threads mas adelante



## Linux: nice



El valor **nice** es una característica del sistema operativo Linux que se usa para ajustar la prioridad de los procesos en cuanto a su planificación para el uso del CPU. En esencia, el valor nice permite a los usuarios manipular la prioridad de un proceso de manera que los procesos con una mayor prioridad obtengan más tiempo de CPU comparado con los que tienen una menor prioridad.

Se puede configurar con la system call `nice()`

# Linux: Funcionamiento del valor nice




**Rango de valores:** El valor nice puede variar de -20 hasta 19. Un valor nice de -20 es el más alto nivel de prioridad (menos "nice", es decir, menos amable con los demás procesos), y 19 es el nivel más bajo (más "nice", más amable). Por defecto, los procesos suelen iniciarse con un valor nice de 0.

**Asignación de CPU:** Un valor nice bajo (más cercano a -20) hace que el proceso sea más "egoísta", obteniendo más tiempo de CPU. Por el contrario, un valor nice alto (más cercano a 19) hace que el proceso sea más "generoso", cediendo el tiempo de CPU a otros procesos.

**Modificación del valor nice:** Los usuarios pueden cambiar el valor nice de un proceso para afectar su prioridad de planificación. Esto se hace generalmente a través de la línea de comandos con herramientas como nice y renice. Por ejemplo, para iniciar un proceso con un valor nice específico, se utiliza el comando nice -n [valor] [comando]. Para cambiar el valor nice de un proceso ya en ejecución, se usa renice [nuevo\_valor] -p [pid].

# Linux: nice

**Pesos de Prioridad (Weight):** Cada proceso tiene un peso asignado basado en su prioridad de nice, que puede variar de -20 (más favorable) a 19 (menos favorable). Estos valores de nice se mapean a un peso específico utilizando una tabla predefinida en el kernel. Los pesos ayudan a determinar *cuán rápido un proceso acumula vruntime en comparación con otros procesos*.

$$\text{vruntime}+ = \frac{\text{delta\_exec}}{\text{weight}}$$


# Linux: Weighting (Niceness) "Weighting (Niceness)" sería Ponderación (Amabilidad/Prioridad)

CFS mapea el nice value con un peso como se muestra :

```
static const int prio_to_weight[40] = {  
    /* -20 */      88761,    71755,    56483,    46273,    36291,  
    /* -15 */      29154,    23254,    18705,    14949,    11916,  
    /* -10 */       9548,     7620,     6100,     4904,     3906,  
    /*  -5 */       3121,     2501,     1991,     1586,     1277,  
    /*   0 */       1024,       820,       655,       526,       423,  
    /*   5 */        335,       272,       215,       172,       137,  
    /*  10 */        110,        87,        70,        56,        45,  
    /*  15 */         36,        29,        23,        18,        15,  
};
```

# Linux: Timeslice dinámico



Ya elegí un proceso. ¿Cuanto tiempo va a ejecutar?

**Latencia objetivo (sched\_latency):** Este es un valor que define el período de tiempo durante el cual el scheduler intenta que todos los procesos ejecutables se ejecuten al menos una vez. Por defecto, este período es de alrededor de 48 ms en sistemas con un número considerable de tareas ejecutándose (el default es 20ms). Este valor se ajusta según el número de tareas activas para mantener el rendimiento del scheduler.

Basicamente, para calcular por cuánto tiempo se va a correr un proceso al ser elegido se toma proporcional del sched latency, la prioridad del proceso dividida por el total de todas las prioridades de todos los procesos runnable.

# Linux: Timeslice dinámico



## Ejemplo

`sched_latency` es de 20 ms:

- Si tengo dos procesos con igual prioridad, el timeslice de cada uno sera de 10 ms.
- Si tengo cuatro procesos con igual prioridad, el timeslice de cada uno sera de 5 ms

# Linux: Timeslice dinámico



## Ejemplo

`sched_latency` es de 20 ms:

- Proceso A: nice-0
- Proceso B: nice-5

nice-0 = 1024

Nice-5 = 335

- $\text{timeslice(A)} = 20 \text{ ms} * 1024 / (1024 + 335) = 15 \text{ ms}$
- $\text{timeslice(B)} = 20 \text{ ms} * 335 / (1024 + 335) = 5 \text{ ms}$

# Linux: Timeslice dinámico



## Ejemplo

`sched_latency` es de 20 ms:

- Proceso A: nice-10
- Proceso B: nice-15

nice-10 = 110

nice-15 = 36

- $\text{timeslice(A)} = 20 \text{ ms} * 110 / (110 + 36) = 15 \text{ ms}$
- $\text{timeslice(B)} = 20 \text{ ms} * 36 / (110 + 36) = 5 \text{ ms}$

El valor de nice tiene sentido relativo. No absoluto!!!



# Linux: Constantes y Factores en el CFS



Mínimo tiempo de ejecución garantizado (**min\_granularity**): Este valor define el mínimo tiempo de ejecución que un proceso debería tener antes de ser desalojado del CPU. Esto evita que el timeslice calculado (diapositiva anterior) sea demasiado pequeño cuando hay muchísimos procesos.

# Linux: Funcionamiento de `sched_latency`



**Interacción con `min_granularity`:** El `sched_latency` trabaja en conjunto con otro parámetro llamado `min_granularity`. Este último establece el tiempo mínimo que un proceso debe ejecutarse antes de ser desalojado del CPU, para evitar que los procesos sean interrumpidos demasiado rápidamente lo cual podría llevar a un aumento en el overhead por cambios de contexto. `min_granularity` suele ser una fracción del `sched_latency`, típicamente alrededor del 10-20%.

## Ejemplo de Ajuste de `sched_latency`

Supongamos que un sistema tiene 100 tareas activas y un `sched_latency` estándar de 48 ms. Si el número de tareas supera el valor ideal que el scheduler puede manejar efectivamente dentro de ese período, digamos 25 tareas, el sistema puede decidir duplicar el `sched_latency` a 96 ms para permitir que todas las tareas sean planificadas adecuadamente sin aumentar demasiado la frecuencia de cambio de contexto.

# Linux: Constantes y Factores en el CFS



**Normalización de tiempos:** Para garantizar que los cálculos se mantengan dentro de límites manejables y evitar el desbordamiento de enteros, los tiempos de ejecución pueden ser normalizados periódicamente. Esto implica ajustar los vruntime de todos los procesos, reduciendo todos los valores en función del proceso con el menor vruntime. Esto evita que haya overflows de las variables.

# Linux: Esquema del Funcionamiento del CFS



## Inicialización

Cada proceso recibe un valor `vruntime` inicial, que es cero cuando el proceso es creado.

## Cálculo de Pesos

Basado en el valor `nice` de cada proceso, se asigna un peso específico.

La tabla de mapeo de `nice` a pesos determina cuán rápido se acumula `vruntime` para un proceso dado.

## Gestión de `vruntime`

El `vruntime` de un proceso se incrementa en función del tiempo que el proceso pasa ejecutándose en el CPU.

La tasa de incremento se ajusta por el peso del proceso: procesos con mayor peso (menos `nice`) acumulan `vruntime` más lentamente.

## Uso de Red-Black Tree

Todos los procesos listos para ejecutarse son almacenados en un árbol rojo-negro, organizados por su `vruntime`.

El proceso con el menor `vruntime` se encuentra siempre el más a la izquierda de la raíz del árbol y es seleccionado para ejecutarse.

# Linux: Esquema del Funcionamiento del CFS



## Selección de Procesos

El scheduler selecciona el proceso con el menor vruntime del árbol para ejecución.

Este proceso se ejecuta hasta que su vruntime ya no es el mínimo, o hasta que se bloquea o termina su quantum de ejecución.

## Preemptividad y Voluntariedad

Si un proceso se bloquea esperando I/O, se retira del árbol temporalmente.

Cuando un proceso despierta o es creado, se calcula su vruntime ajustado y se inserta en el árbol.

## Ajuste Dinámico

`sched_latency` y `min_granularity` se utilizan para ajustar la cantidad de tiempo que cada proceso puede ejecutarse.

En sistemas con muchos procesos, estos valores se ajustan para balancear equidad y eficiencia.

## Balanceo de Carga entre CPUs

El CFS también maneja la distribución de procesos entre múltiples CPUs.

Los procesos pueden ser migrados entre CPUs para optimizar la carga y minimizar la latencia.

# Linux: Ejemplo



## Detalles de los Procesos y Valores Nice

Proceso A: nice = 0

Proceso B: nice = -5

Proceso C: nice = 10

Proceso D: nice = 0

Proceso E: nice = -10

# Linux: Cálculo de Pesos



El peso de un proceso se basa en su valor nice. En Linux, los pesos están predefinidos y se pueden ver en el código fuente del kernel. Para este ejemplo, vamos a suponer algunos valores hipotéticos para los pesos:

Valor nice de -10 (Proceso E) corresponde a un peso de 1024.

Valor nice de -5 (Proceso B) corresponde a un peso de 512.

Valor nice de 0 (Proceso A y D) corresponde a un peso de 256.

Valor nice de 10 (Proceso C) corresponde a un peso de 128.

# Linux



Inicio: Todos los procesos empiezan con  $\text{vruntime} = 0$ .

## Ciclo 1:

Todos los procesos tienen el mismo  $\text{vruntime}$  inicial, pero el Proceso E tiene el peso más alto y, por lo tanto, es el menos penalizado en la acumulación de  $\text{vruntime}$ .

El Proceso E se ejecuta, digamos, por una unidad de tiempo. Su  $\text{vruntime}$  incrementa menos debido a su alto peso.

Nuevo  $\text{vruntime}$  de E =  $0 + (1 \text{ unidad de tiempo} / 1024) = 0.0009765625$ .

## Ciclo 2:

El Proceso B tiene el siguiente peso más alto. Se ejecuta por una unidad de tiempo.

Nuevo  $\text{vruntime}$  de B =  $0 + (1 \text{ unidad de tiempo} / 512) = 0.001953125$ .

## Ciclo 3:

Proceso A o D (ambos tienen  $\text{nice} = 0$  y el mismo peso).

Supongamos que se selecciona el Proceso A. Se ejecuta por una unidad de tiempo.

Nuevo  $\text{vruntime}$  de A =  $0 + (1 \text{ unidad de tiempo} / 256) = 0.00390625$ .



# Linux



## Ciclo 4:

Se ejecuta el Proceso D bajo las mismas condiciones que A.

Nuevo vruntime de D =  $0 + (1 \text{ unidad de tiempo} / 256) = 0.00390625$ .

## Ciclo 5:

El Proceso C tiene el peso más bajo y por lo tanto, su vruntime crece más rápidamente. Se ejecuta por una unidad de tiempo.

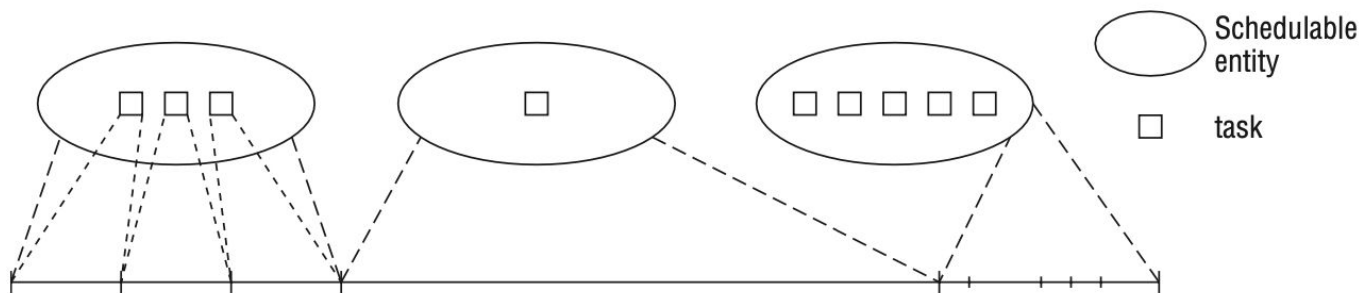
Nuevo vruntime de C =  $0 + (1 \text{ unidad de tiempo} / 128) = 0.0078125$ .

## Revisión y Selección Continua:

Al final de estos ciclos, el scheduler revisa todos los vruntime. El Proceso E, que ha acumulado el menor vruntime (0.0009765625), sería normalmente seleccionado nuevamente, ya que su vruntime sigue siendo el más bajo.

# Linux: Group Scheduling

Los procesos se colocan en diferentes grupos, y el planificador es primero justo entre estos grupos y luego justo entre todos los procesos dentro del grupo.



**Figure 2-29: Overview of fair group scheduling: The available CPU time is first distributed fairly among the scheduling groups, and then between the processes in each group.**

# Linux: Group Scheduling



Esto permite, por ejemplo, otorgar partes iguales del tiempo de CPU disponible a cada usuario. Una vez que el planificador ha decidido cuánto tiempo obtiene cada usuario, el intervalo determinado se distribuye de manera justa entre los procesos del usuario. Esto implica naturalmente que, cuanto más procesos ejecute un usuario, menor será el share de CPU que obtendrá cada proceso. Sin embargo, la cantidad de tiempo total para el usuario no se ve influenciada por el número de procesos.

# Linux



**Cgroups y control de recursos:** El kernel también ofrece grupos de control (control groups), que permiten (a través de un filesystem especial cgroups), crear colecciones arbitrarias de tareas, que incluso pueden ordenarse en múltiples jerarquías.

# Linux

The Linux logo, which consists of a horizontal bar divided into four segments of different colors: teal, orange, yellow, and blue.

El otro gran uso de grupos de scheduler son los threads. Los threads se agrupan en un grupo que representa al proceso...

... pero ¿qué son los threads?

---

# Threads

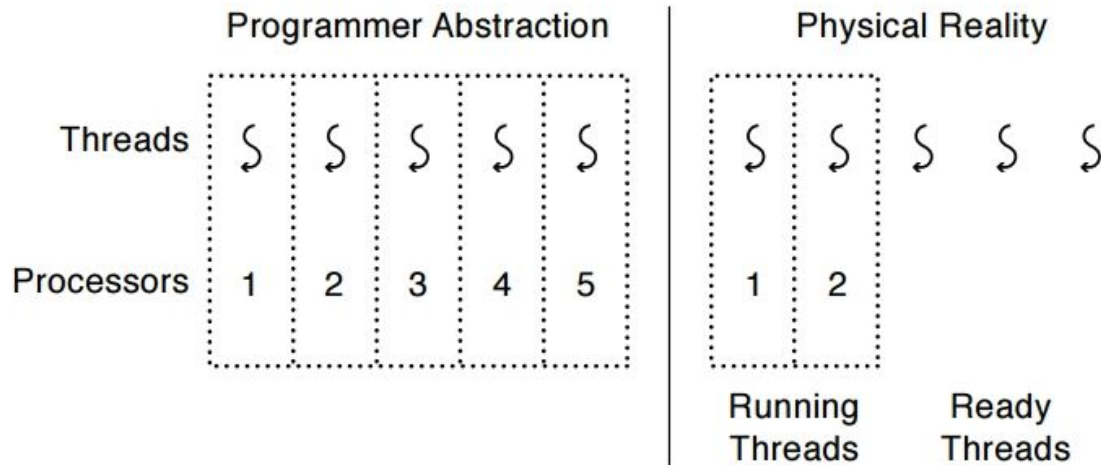
(y su relación con los procesos)

## Conceptos clave:

- Definición de threads
- Thread scheduler
- Interleaving (entrelazamiento)

# A qué apuntamos

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 muy precisa. El concepto básico es el siguiente:





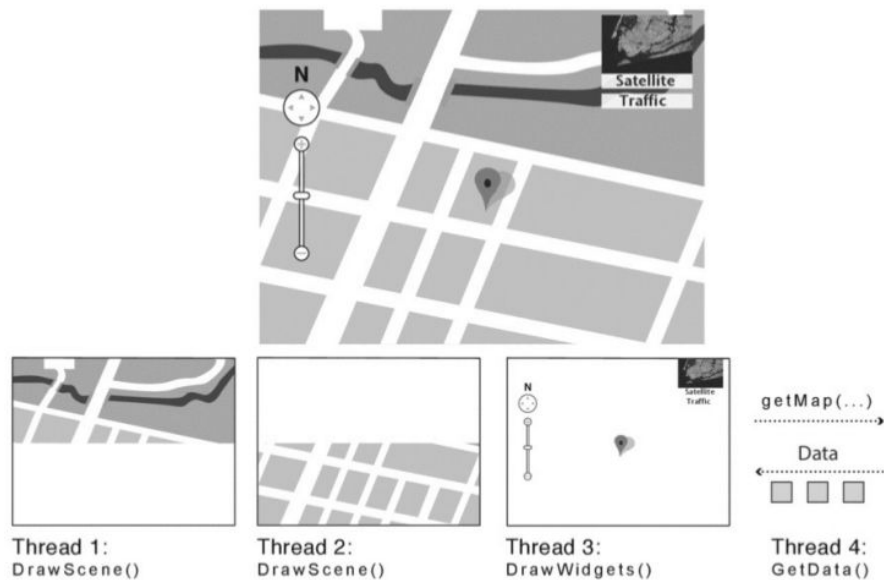
# La Abstraccion

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 sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él desee.



# La Abstraccion





## Las dificultades de la concurrencia

- Inteligibilidad
- Predecibilidad
- No-Determinismo

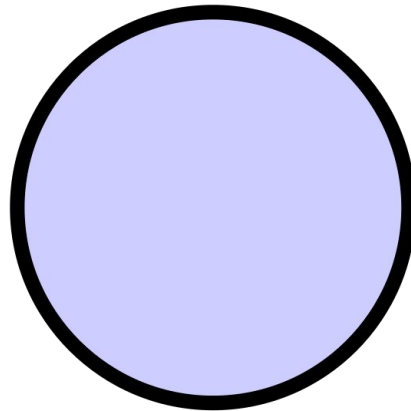


## Threads vs procesos

**Proceso:** un programa en ejecución con derechos restringidos.

**Thread:** una secuencia independiente de instrucciones ejecutándose dentro de un programa.

### Process



### Thread





# Threads

Esta abstracción, el thread, se caracteriza por :

- Thread id
- un conjunto los valores de registros
- stack propio
- una política y prioridad de ejecución
- un propio errno
- datos específicos del thread

Thread

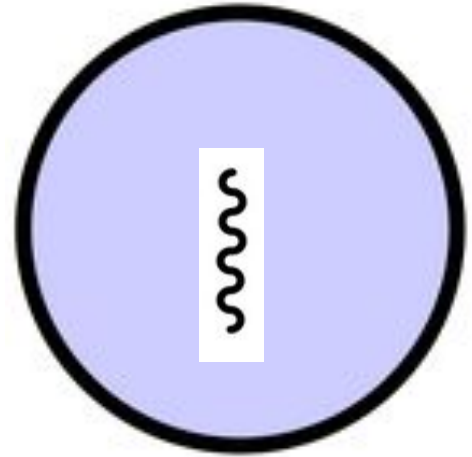




## Threads

**One thread per process:** un proceso con una única secuencia de instrucciones ejecutándose de inicio a fin.

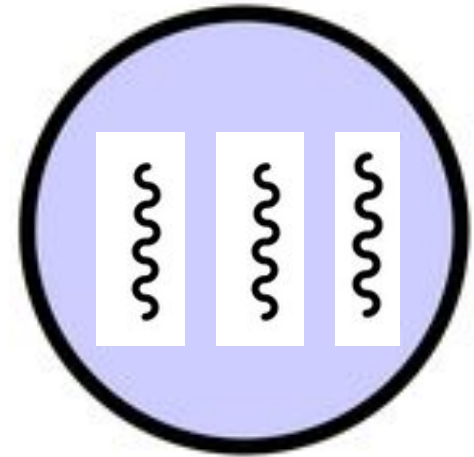
## Process

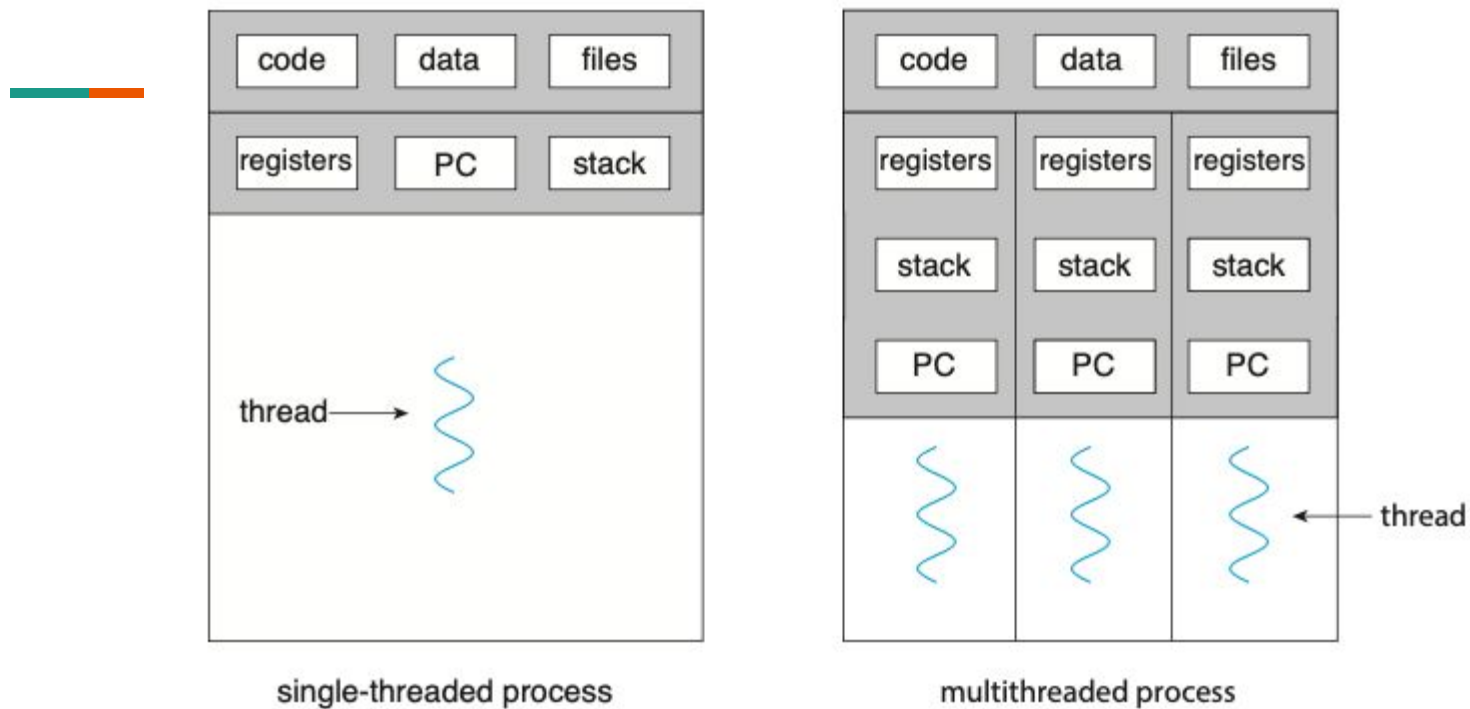


## Threads

- Many threads per process: un programa es visto como threads ejecutándose dentro de un proceso con derechos restringidos. En un determinado instante, algunos threads pueden estar corriendo y otros estar suspendidos. Cuando se detecta por ejemplo una operación de I/O por alguna interrupción, el kernel desaloja (preempt) a algunos de los threads que están corriendo, atiende la interrupción, y al terminar de manejar la interrupción vuelve a correr el thread nuevamente.

Process





**Figure 4.1** Single-threaded and multithreaded processes.



## Thread Scheduler

¿Cómo hace el S.O. para crear la ilusión de muchos threads con un número fijo de procesadores?

Obviamente es necesario un planificador de thread o **threads scheduler**, ya que el S.O. podría estar trabajando con un único procesador. El cambio entre threads es transparente, es decir que el programador debe preocuparse de la secuencia de instrucciones y no el cuando éste debe ser suspendido o no.

Por ende los Threads **proveen un modelo de ejecución en el cual cada thread corre en un procesador virtual dedicado (exclusivo) con una velocidad variable e impredecible**  
Anderson-Dahlin, pag 138.





## Thread Scheduler planificador de hilos

Esto quiere decir que desde el punto de vista del thread cada instrucción se ejecuta inmediatamente una detrás de otra. Pero el que decide cuando se ejecuta es el planificador de threads o thread scheduler. Por ejemplo:

```
...  
...  
x = x + 1;  
y = x + y;  
z = x + 5y;  
...  
...
```

# Thread Scheduler

Entonces  
base a  
antedicho,  
pueden  
encontrar  
siguientes  
escenarios  
ejecución:

## Programmer's View

.  
.  
.  
x = x + 1;  
y = y + x;  
z = x + 5y;  
.  
.  
.

## Possible Execution #1

.  
.  
.  
x = x + 1;  
y = y + x;  
z = x + 5y;  
.  
.  
.

## Possible Execution #2

.  
.  
.  
x = x + 1;  
.....  
Thread is suspended.  
Other thread(s) run.  
Thread is resumed.  
.....  
y = y + x;  
z = x + 5y;

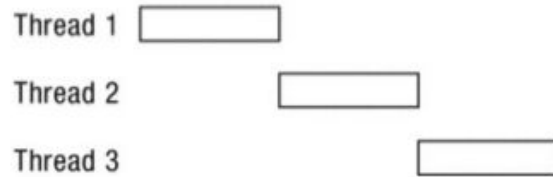
## Possible Execution #3

.  
.  
.  
x = x + 1;  
y = y + x;  
.....  
Thread is suspended.  
Other thread(s) run.  
Thread is resumed.  
.....  
z = x + 5y;

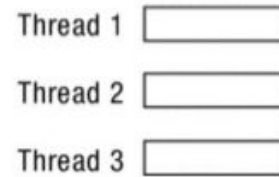
# Thread Scheduler

Y los  
siguientes  
interleaves o  
entrelazado  
pueden  
suceder, con  
estos distintos  
threads

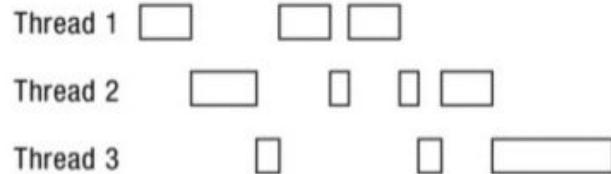
One Execution



Another Execution



Another Execution





# Thread Scheduler

En la actualidad hay dos formas de que los threads se relacionan entre sí:

- Multi-threading Cooperativo: no hay interrupción a menos que se solicite.  
→ ¿Problemas?
- Multi-threading Preemptivo: Es el más usado en la actualidad. Consiste en que un threads en estado de running puede ser movido en cualquier momento.

---

# API de Threads

## Conceptos clave:

- Biblioteca pthreads
- Creación y terminación de threads



## Creacion

**thread:** Es un puntero a la **estructura** de tipo **pthread\_t**, que se utiliza para interactuar con el threads.

**attr:** Se utiliza para especificar los ciertos atributos que el thread deberia tener, por ejemplo, el tamaño del stack, o la prioridad de scheduling del thread. En la mayoría de los casos es NULL.

**start\_routine:** Sea tal vez el argumento más complejo, pero no es más que un puntero a una función, en este caso que devuelve void.

**arg:** Es un puntero a void que debe apuntar a los argumentos de la función.

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




## Terminación de un thread

Muchas veces es necesario esperar a que un determinado thread finalice su ejecución, para ello se utiliza la función `pthread_join()`, que toma dos argumentos

1. `thread` es el thread por el que hay que esperar y es de tipo `pthread_t`.
2. `value_ptr` es el puntero al valor esperado de retorno.

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



```
#include <pthread.h>
#include <stdio.h>

void *mythread(void *arg) {

    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;

    printf("inicia main() \n");
    rc=pthread_create(&p1, NULL, mythread, "A");
    rc=pthread_create(&p2, NULL, mythread, "B");

    rc=pthread_join(p1, NULL);
    rc=pthread_join(p2, NULL);

    printf("termina main() \n");
    return 0;
}
```





## Tabla de equivalencia entre procesos y threads:

| Process primitive | Thread primitive | Description  |
|-------------------|------------------|--|
| fork              | pthread_create   | crea un nuevo flujo de control                                       |
| exit              | pthread_exit     | sale de un flujo de control existente                                |
| waitpid           | pthread_join     | obtiene el estado de salida de un flujo de control                   |
| atexit            | pthread_cleanup  | función a ser llamada en el momento de salida de un flujo de control |
| getpid            | pthread_self     | obtiene el id de un determinado flujo de control                     |
| abort             | pthread_cancel   | terminación anormal de un flujo de control                           |

---

# Threads: implementación

## Conceptos clave:

- Estructuras del SO para soportar threads
- Estados
- Implementación en Linux
- clone, fork y vfork
- Copy on Write



## Threads: Estructura

Como se ha visto, cada thread es la representación de una secuencia de ejecución de un conjunto de instrucciones. El S.O. provee la ilusión de que cada uno de estos threads se ejecutan en su propio procesador, haciendo de forma transparente que se ejecuten o paren su ejecución.

Para que la ilusión sea creíble, el sistema operativo debe guardar y cargar el estado de cada thread. Como cualquier thread puede correr en el procesador o en el kernel, también debe haber estados compartidos, que no deberían cambiar entre los modos.

Para poder entender la abstracción hay que comprender que existen dos estados:

- El estado per thread.

- El estado compartido entre varios threads.

# El Estado Per-thread y Threads Control Block (TCB)



Cada thread debe tener una estructura que represente su estado. Esta estructura se denomina Thread Control Block (TCB), se crea una entrada por cada thread. La TCB almacena el estado per-thread de un thread:

El estado del Cómputo que debe ser realizado por el thread.

Para poder crear múltiples threads y pararlos y arrancarlos, el S.O. debe poder almacenar en la TCB el estado actual del bloque de ejecución:

El puntero al stack del thread.

Una copia de sus registros en el procesador.



## Metadata del thread

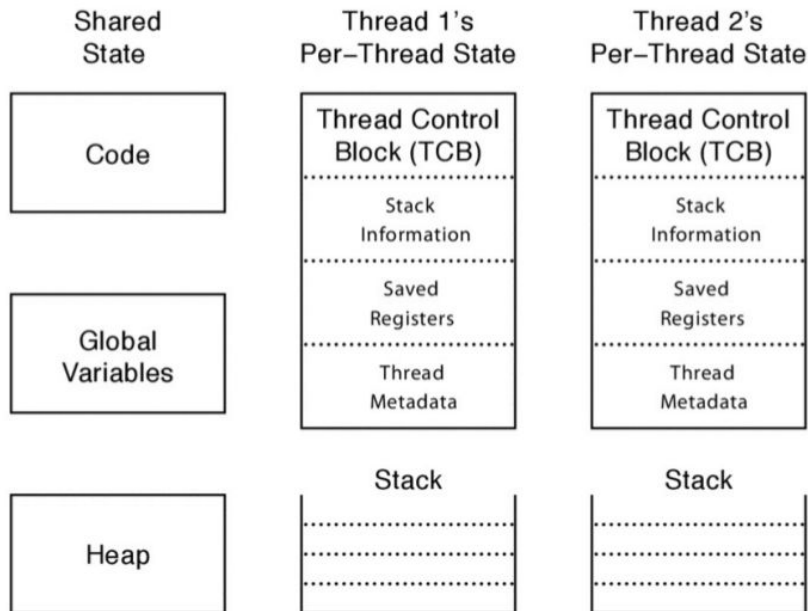
Por cada thread se debe guardar determinada información sobre el mismo:

- ID
- Prioridad de scheduling
- Status

# Metadata del thread

De forma contraria al per-thread state se debe guardar cierta información que es compartida por varios Threads:

- El Código
- Variables Globales
- Variables del Heap



# Memoria en un proceso multithread

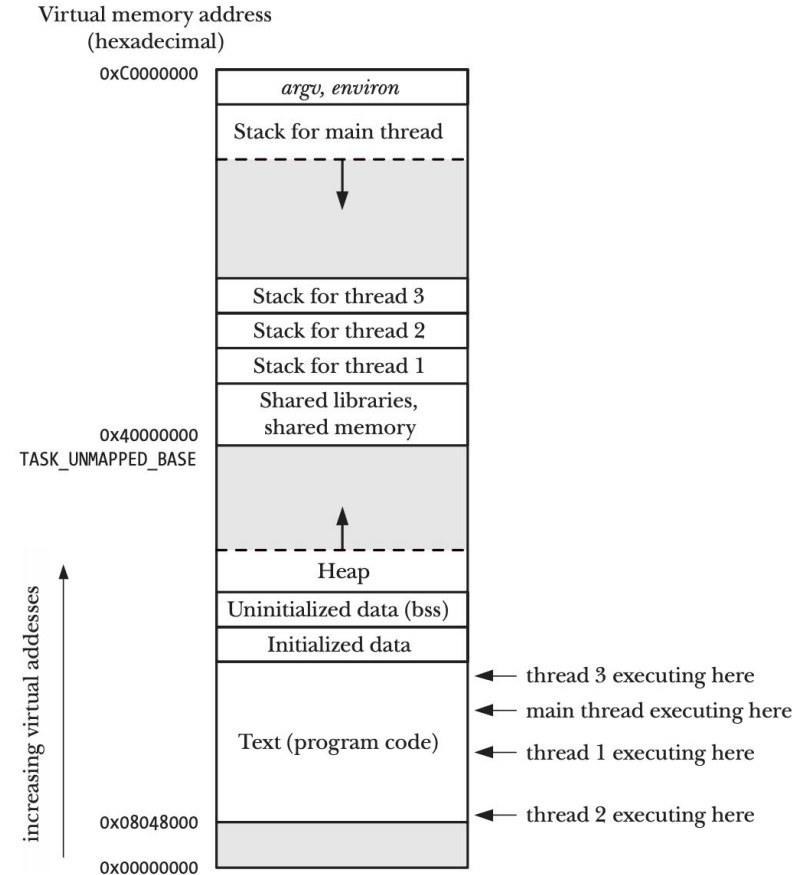
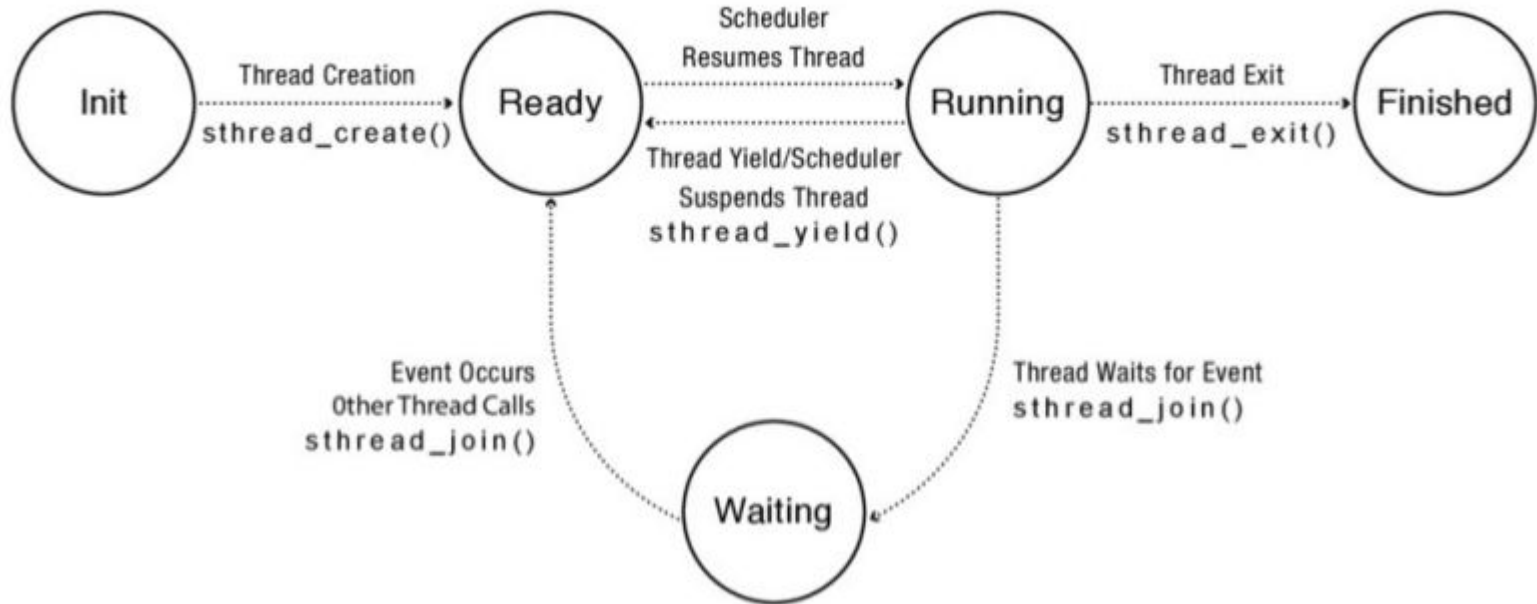


Figure 29-1: Four threads executing in a process (Linux/x86-32)

# Estados







## Estados

**Init:** Un thread se encuentra en estado INIT mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras. Una vez que esto se ha realizado el estado del thread se setea en READY. Además se lo pone en una lista llamada ready list en la cual están esperando todos los thread listos para ser ejecutados en el procesador.

**Ready:** Un thread en este estado está listo para ser ejecutado pero no está siendo ejecutado en ese instante. La TCB está en la ready list y los valores de los registros están en la TCB. En cualquier momento el thread scheduler puede transicionar al estado RUNNING.



## Estados

**Running:** Un thread en este estado está siendo ejecutado en este mismo instante por el procesador. En este mismo instante los valores de los registros están en el procesador. En este estado un RUNNING THREAD puede pasar a READY de dos formas:

- El scheduler puede pasar un thread de su estado RUNNING a READY mediante el desalojo o preemption del mismo mediante el guardado de los valores de los registros y cambiando el thread que se está ejecutando por el próximo de la lista.
- Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de `thread_yield`, por ejemplo.



## Estados

**Waiting:** En este estado el Thread está esperando que algún determinado evento suceda. Dado que un thread en WAITING no puede pasar a RUNNING directamente, estos thread se almacenan en la lista llamada waiting list. Una vez que el evento ocurre el scheduler se encarga de pasar el thread del estado WAITING a RUNNING, moviendo la TCB desde el waiting list a la ready list.

**Finished:** Un thread que se encuentra en estado FINISHED nunca más podrá volver a ser ejecutado. Existe una lista llamada finished list en la que se encuentran las TCB de los threads que han terminado.



# Diferencias Proceso/Thread

## Los threads

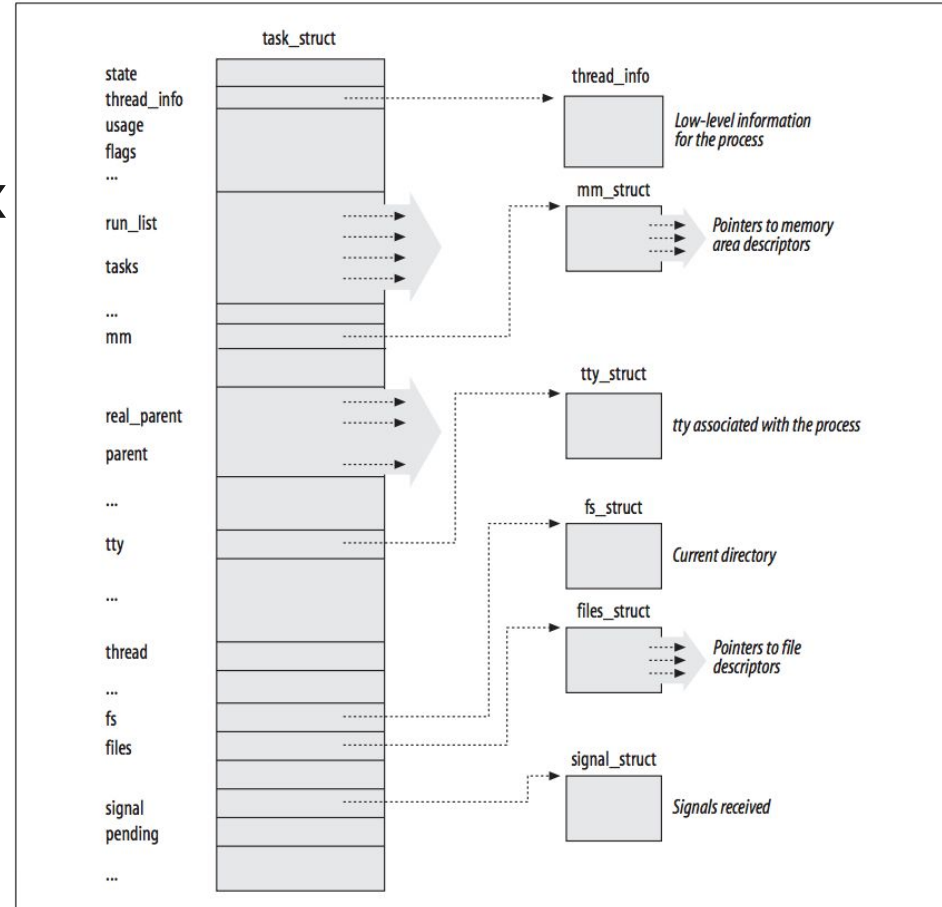
- Por defecto comparten memoria
- Por defecto comparten los descriptores de archivos
- Por defecto comparten el contexto del filesystem
- Por defecto comparten el manejo de señales

## 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

# Modelo de thread en linux

Linux utiliza un modelo 1-1 (proceso-thread), con lo cual dentro del kernel no existe distinción alguna entre thread y proceso – todo es un tarea ejecutable.





# Creación de procesos y threads en Linux

## Creacion de Threads (LinuxThreads)

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

## fork()

```
clone(SIGCHLD, 0);
```

## vfork()

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```



## **vfork**

(Berkeley Software Distribution)

Las primeras implementaciones de BSD estuvieron entre aquellas en las que `fork()` realizaba una duplicación literal de los datos, heap y stack del proceso padre.

Esto es ineficiente, especialmente si el `fork()` es seguido por un `exec()` inmediato. Por esta razón, versiones posteriores de BSD introdujeron la llamada al sistema `vfork()`, que era mucho más eficiente que el `fork()` de BSD, aunque operaba con una semántica ligeramente diferente (de hecho, algo extraña).



## vfork

Al igual que `fork()`, `vfork()` es utilizado por el proceso que lo llama para crear un nuevo proceso hijo. Sin embargo, `vfork()` está diseñado específicamente para ser usado en programas donde el hijo realiza una llamada `exec()` inmediata.

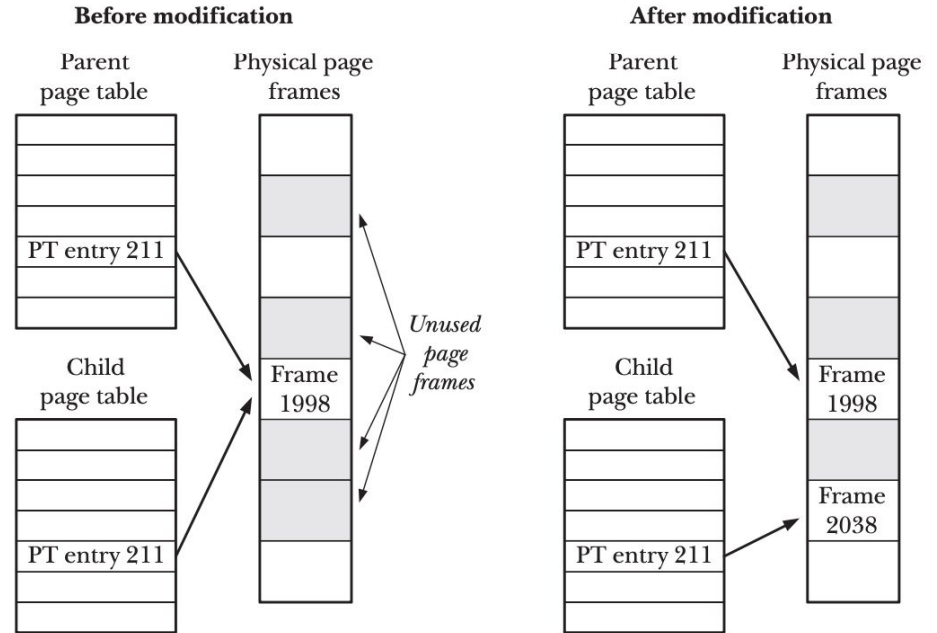
Dos características distinguen la llamada al sistema `vfork()` de `fork()` y la hacen más eficiente:

- No se duplica ninguna página de memoria virtual ni tablas de páginas para el proceso hijo. En su lugar, el hijo comparte la memoria del padre hasta que realiza un `exec()` exitoso o llama a `_exit()` para terminar.
- La ejecución del proceso padre se suspende hasta que el hijo haya realizado un `exec()` o `_exit()`.



# COW: Copy on write

Inicialmente, el kernel configura las cosas de manera que las entradas de la tabla de páginas para estos segmentos se refieren a las mismas páginas de memoria física que las entradas correspondientes de la tabla de páginas del padre, y las páginas en sí mismas se marcan como de solo lectura.



**Figure 24-3:** Page tables before and after modification of a shared copy-on-write page

# COW: Copy on write

Después del `fork()`, el kernel intercepta cualquier intento del padre o del hijo de modificar una de estas páginas y hace una copia duplicada de la página que está a punto de ser **modificada**. Esta nueva copia de la página se asigna al proceso que causó el fault, y la entrada correspondiente de la tabla de páginas para el hijo se ajusta de manera apropiada. A partir de este punto, el padre y el hijo pueden modificar sus copias privadas de la página sin que los cambios sean visibles para el otro proceso.

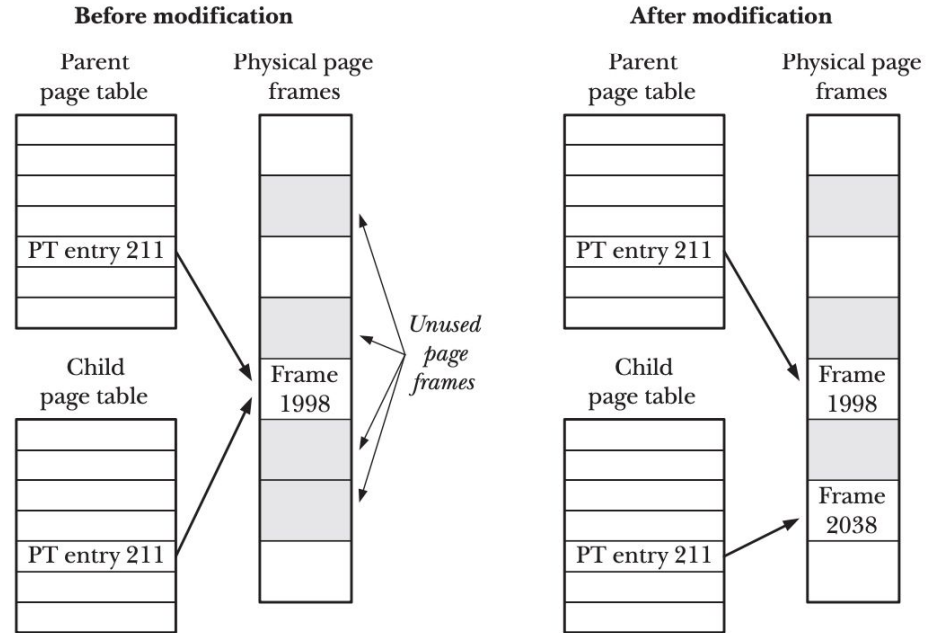


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

# COW

## Implementacion en RISC-V

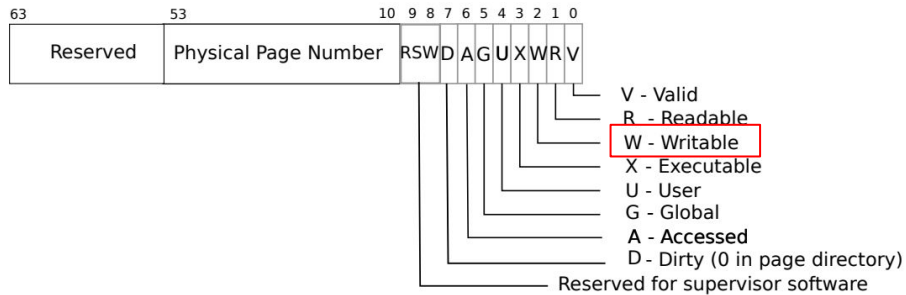


Figure 3.2: RISC-V address translation details.

# Velocidad de creación de procesos

**Table 28-3:** Time required to create 100,000 processes using *fork()*, *vfork()*, and *clone()*

| Method of process creation     | Total Virtual Memory |       |                   |       |                   |       |
|--------------------------------|----------------------|-------|-------------------|-------|-------------------|-------|
|                                | 1.70 MB              |       | 2.70 MB           |       | 11.70 MB          |       |
|                                | Time (secs)          | Rate  | Time (secs)       | Rate  | Time (secs)       | Rate  |
| <i>fork()</i>                  | 22.27<br>(7.99)      | 4544  | 26.38<br>(8.98)   | 4135  | 126.93<br>(52.55) | 1276  |
| <i>vfork()</i>                 | 3.52<br>(2.49)       | 28955 | 3.55<br>(2.50)    | 28621 | 3.53<br>(2.51)    | 28810 |
| <i>clone()</i>                 | 2.97<br>(2.14)       | 34333 | 2.98<br>(2.13)    | 34217 | 2.93<br>(2.10)    | 34688 |
| <i>fork()</i> + <i>exec()</i>  | 135.72<br>(12.39)    | 764   | 146.15<br>(16.69) | 719   | 260.34<br>(61.86) | 435   |
| <i>vfork()</i> + <i>exec()</i> | 107.36<br>(6.27)     | 969   | 107.81<br>(6.35)  | 964   | 107.97<br>(6.38)  | 960   |



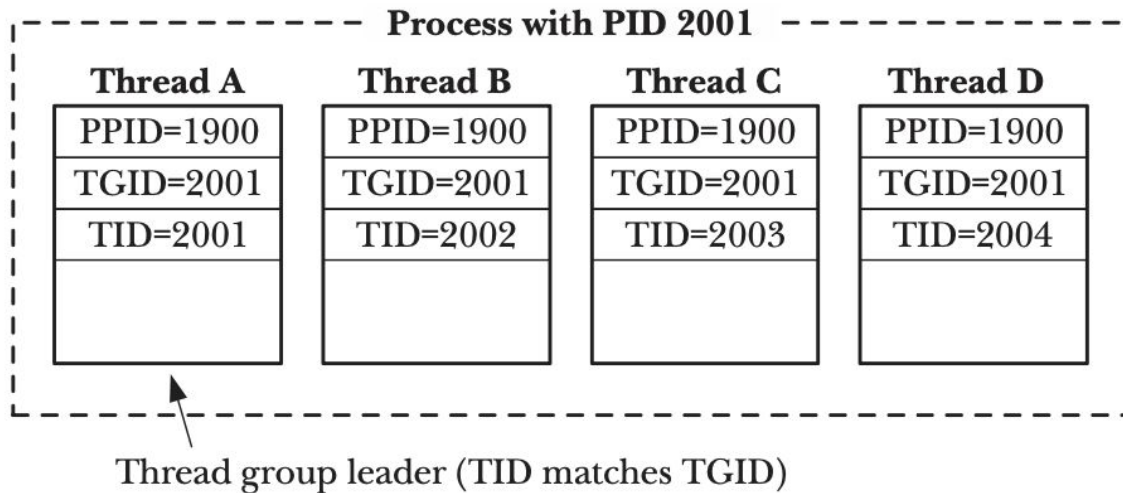
## Native POSIX Threads Library (NPTL)

```
clone(CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |  
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |  
CLONE_CHILD_CLEARTID | CLONE_SYSVSEM, 0)
```

Se puede comprobar si se está usando en el sistema ejecutando:

```
$ /lib/libc.so.6
```

## Thread Groups: CLONE\_THREAD



**Figure 28-1:** A thread group containing four threads



## Viendo los Threads: ps m

---

```
$ ps m
  PID TTY          STAT TIME  COMMAND
 3587 pts/3    -      0:00 bash❶
    - -      Ss      0:00 -
 3592 pts/4    -      0:00 bash❷
    - -      Ss      0:00 -
12534 tty7      -    668:30 /usr/lib/xorg/Xorg -core :0❸
    - -      Ssl+    659:55 -
    - -      Ssl+     0:00 -
    - -      Ssl+     0:00 -
    - -      Ssl+     8:35 -
```

---

*Listing 8-1: Viewing threads with ps m*

This listing shows processes along with threads. Each line with a number in the PID column (at ❶, ❷, and ❸) represents a process, as in the normal ps output. The lines with dashes in the PID column represent the threads associated with the process. In this output, the processes at ❶ and ❷ have only one thread each, but process 12534 at ❸ is multithreaded, with four threads.

## Viendo los Threads: ps -T

```
prateekjangid@prateekjangid:~$ ps -T -p 1904
```

| PID  | SPID | TTY | TIME     | CMD       |
|------|------|-----|----------|-----------|
| 1904 | 1904 | ?   | 00:00:00 | gjs       |
| 1904 | 1926 | ?   | 00:00:00 | gmain     |
| 1904 | 1932 | ?   | 00:00:00 | gdbus     |
| 1904 | 1940 | ?   | 00:00:00 | JS Helper |
| 1904 | 1941 | ?   | 00:00:00 | JS Helper |

```
prateekjangid@prateekjangid:~$
```



---

# Bibliografía

# Bibliografía

## Completely Fair Scheduler

- **Linux kernel development** / Robert Love. — 3rd ed.
  - Capítulo 4 - Process Scheduling
- **Professional Linux kernel architecture** / Wolfgang Mauerer.
  - Chapter 2: Process Management and Scheduling

## Threads

- **Operating Systems Principles and Practice, Volume 2: Concurrency** / Michael Dahlin, Thomas Anderson
  - Chapter 4 - Concurrency and Threads
- **The Linux programming interface : a Linux and UNIX system programming handbook** / by Michael Kerrisk.
  - Chapter 24: Process Creation
  - Chapter 29 Threads:introduction

## Simulador de scheduler

<https://github.com/fisop/sched-sim>