Manejo y Planificación de Procesos

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
	2016-05-06		CYF

Contents

Los Procesos

Un proceso contiene:

- El código objeto: segmento *text*
- Lista de descriptores de archivos abiertos
- Señales pendientes
- Estructura del proceso para el kernel
- Estado
- Direcciones de memoria usadas
- Hilos de ejecución
- Variables globales: segmento *data*

Un hilo puede contener:

- Contadores de ejecución
- Comparte la pila del proceso del padre y otros hilos
- Registros de procesador en uso

En el kernel, todo es un hilo. Un proceso es un hilo. Un hilo es un hilo.

Para el programa, el kernel le presenta un *Procesador Virtual* y *Memoria Virtual*.

El *Procesador Virtual* es administración del planificador de procesos/hilos.

La *Memoria Virtual* es administrada por el gestor de memoria.

Los hilos comparten la *Memoria Virtual*, pero no el *Procesador Virtual*.

Los hilos comunes podrían compartir la tabla de descriptores de archivos y espacio de direccionamiento.

Los "procesos" son creados a través de la llamada al sistema *fork()* (que en Linux en realidad se llama *clone()*). *fork()* duplica el estado de un proceso (el padre) a través de *Copy on Write*.

La llamada al sistema *exec()* crea segmento y direccionamiento nuevo para el nuevo proceso, si es que es creado a través de esta llamada al sistema.

El proceso acaba llamandose a sí mismo con la llamada *exit()* (que en realidad se hacen otras llamadas como *wait()* o *waitpid()* antes de que *exit()* llegue por fin).

Dentro del código fuente del Kernel, el término usado para los hilos es en realidad `tareas` (*task*).

Descriptores de Procesos y Estructuras de Tareas

Descriptor de Proceso

La lista de procesos es guardada en una lista doble enlazada circular llamada *task list*. Cada elemento en la lista es un descriptor de proceso del tipo `struct task_struct`, definida en <*linux/sched.h*>.

En esta estructura se almacena toda la información concerniente a un proceso, archivos abiertos, su espacio de direccionamiento, señales pendientes, su estado, etcétera.

```
struct task_struct {
    volatile long state;
    void *stack;
    atomic_t usage;
    unsigned int flags;
    unsigned int ptrace;
    int on_rq;
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    struct sched_dl_entity dl;
    unsigned int policy;
    int nr_cpus_allowed;
    cpumask_t cpus_allowed;
    struct list_head tasks;
    struct mm_struct *mm, *active_mm;
    u32 vmacache_seqnum;
    struct vm_area_struct *vmacache[VMACACHE_SIZE];
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;
    unsigned long jobctl;
    unsigned int personality;
    unsigned sched_reset_on_fork:1;
    unsigned sched_contributes_to_load:1;
    unsigned sched_migrated:1;
    unsigned :0;
    unsigned in_execve:1;
    unsigned in_iowait:1;
    unsigned long atomic_flags;
    struct restart_block restart_block;
    pid_t pid;
    pid_t tgid;
    struct task_struct __rcu *real_parent;
    struct task_struct __rcu *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;
    struct list_head ptraced;
    struct list_head ptrace_entry;
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;
    struct list_head thread_node;
    struct completion *vfork_done;
    int __user *set_child_tid;
    int __user *clear_child_tid;
    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    struct prev_cputime prev_cputime;
    unsigned long nvcsw, nivcsw;
    u64 start_time;
    u64 real_start_time;
    unsigned long min_flt, maj_flt;
    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];
    const struct cred __rcu *real_cred;
    const struct cred __rcu *cred;
    char comm[TASK_COMM_LENN];
    struct nameidata *nameidata;
```

```
    struct fs_struct *fs;
    struct files_struct *files;
    struct nsproxy *nsproxy;
    struct signal_struct *signal;
    struct sighand_struct *sighand;
    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask;
    struct sigpending pending;
    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    struct callback_head *task_works;
    struct audit_context *audit_context;
    struct seccomp seccomp;
    u32 parent_exec_id;
    u32 self_exec_id;
    spinlock_t alloc_lock;
    raw_spinklock_t pi_lock;
    struct wake_q_node wake_q;
    void *journal_info;
    struct bio_list *bio_list;
    struct reclaim_state *reclaim_state;
    struct backing_dev_info *backing_dev_info;
    struct io_context *io_context;
    unsigned long ptrace_message;
    siginfo_t *last_siginfo;
    struct task_io_accounting ioac;
    struct rcu_head rcu;
    struct pipe_inode_info *splice_pipe;
    struct page_frag task_frag;
    int nr_dirtied;
    int nr_dirtied_pause;
    unsigned long dirty_paused_when;
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;
    int pagedefault_disabled;
    struct thread_struct thread;
};
```

La estructura `task_struct` es asignada por *SLAB* para proporcionar reuso de objetos generados y Cache.

Al final de la pila del proceso, se almacena una estructura `thread_info`, la cual tiene la ubicación de `task_struct` en uso, escrita en: *arch/arm/include/asm/thread_info.h*

```
struct thread_info {
    unsigned long flags;
    int preempt_count;
    mm_segment_t addr_limit;
    struct task_struct *task;
    __u32 cpu;
    __u32 cpu_domain;
    struct cpu_context_save cpu_context;
    __u32 syscall;
    __u8 used_cp[16];
    unsigned long tp_value[2];
    union fp_state fpstate __attribute__((aligned(8)));
    union vfp_state vfpstate;
};
```

El Identificador de Proceso *PID* se mantiene por compatibilidad de versiones anteriores y por legado UNIX, este valor se guarda en una variable dinámica, controlada por el mismo sistema de hilos del Kernel.

Teóricamente, el kernel de Linux puede soportar 4 millones de procesos en ejecución, antes, el número máximo era de 32768.

El número máximo de procesos puede ser controlado en ejecución por medio de */proc/sys/kernel/pid_max*.

El puntero a `task_struct` del proceso en ejecución es guardado en el registro **r13**, y se actualiza cada vez que hay un cambio de contexto.

Estados de Procesos

Los procesos pueden encontrarse en diferentes estados.

- `TASK_RUNNING`: En ejecución o en cola para ser ejecutado, el proceso podría estar en espacio de usuario (función en *LibC*) o en espacio de kernel (función en *syscall*).
- `TASK_INTERRUPTIBLE`: Proceso dormido o bloqueado, en espera de alguna condición. Si la condición ha sido recibida cambia a `TASK_RUNNING` para ponerlo en cola de ejecución.
- `TASK_UNINTERRUPTIBLE`: Similar al estado anterior, pero no puede recibir ninguna señal, usado en tareas en proceso de bifurcación.
- `_TASK_TRACED`: En estado monitoreado por *ptrace*.
- `_TASK_STOPPED`: Proceso detenido, ni tampoco podrá entrar en ejecución, con señal `SIGSTOP`.

Manipulación de los Estados de Procesos

En espacio de kernel, se tiene que definir manualmente el estado de un proceso (por ejemplo, un hilo de kernel), usando:

```
#include <linux/sched.h>

set_task_state(tarea, estado);

set_current_state(estado);
```

El proceso puede estar en los diferentes contextos:

- **Espacio de Usuario**: El proceso está en periodo de ejecución haciendo algún procedimiento que no requiera una llamada al sistema (operación aritmética).
- Espacio de Kernel, **Contexto de Proceso**: El proceso entra a espacio de kernel cuando hace una llamada al sistema, y sale de cola de ejecución.

Al terminar la llamada al sistema (el kernel sale de ejecución), el siguiente proceso a ejecutar es el próximo en cola de ejecución, no necesariamente el proceso que hizo la llamada.

Dentro de `task_struct` se tienen apuntadores a los procesos padres e hijos (si los hay). El proceso 1 se guarda estáticamente en `init_task`.

Creación de Procesos

Copy on Write

En Linux, los procesos padre e hijos pueden compartir una copia del espacio de direcciones, y las modificaciones del hijo empiezan a escribirse en un nuevo lugar, mientras los datos del padre se mantienen en solo lectura.

Esta técnica permite la demora de copiar cada página de espacio de direcciones al hijo hasta cuando este empieza a escribir; como ventaja es que la bifurcación de procesos es más rápida, ahorrando al mismo tiempo espacio de direccionamiento.

Bifurcación

En Linux, `fork()` no existe, en su lugar se usa `clone()` como llamada al sistema. En la biblioteca de C, se mantiene la función `fork()` por compatibilidad con POSIX.

El proceso de `clone()` es como sigue:

1. Efectúa una llamada a `do_fork()`
2. `do_fork()` llama a `copy_process()`.
3. `copy_process()` llama a `dup_task_struct()`, que crea una nueva pila, una estructura `thread_info`, y `task_struct` para el nuevo proceso, de momento son las mismas estructuras que el padre.
4. Se revisa que el nuevo hijo esté en el límite de procesos permitidos para el usuario.
5. Se actualizan algunas partes de las estructuras de procesos para diferenciar al hijo.
6. Se cambia a estado `TASK_UNINTERRUPTIBLE` para evitar que entre a cola de ejecución.
7. `copy_process()` llama a `copy_flags()` para actualizar valores de `task_struct`.
8. Se llama a `alloc_pid()` para asignarle un *PID*.
9. `copy_process()` duplica o comparte archivos abiertos, información del sistema de archivos, señales, espacio de direccionamiento y espacio con nombre. Entre hilos esta información es compartida, para el hijo, es copiada.
10. `copy_process()` limpia y regresa un puntero al hijo.
11. Si `copy_process()` es exitoso, el hijo es despertado y se envía a cola de ejecución.

El kernel ejecuta los hijos recién creados primero, teóricamente.

`vfork()` - 3BSD

Es una llamada al sistema especial, similar a la llamada de biblioteca `fork()`, pero las entradas de tabla de páginas del padre no son copiadas. El hijo se ejecuta como un hilo único en el espacio de direccionamiento del padre, y el padre es bloqueado hasta que el hijo llame a `exec()` o termine.

El hijo no tiene permiso de escribir en el espacio de direcciones.

Esta llamada al sistema todavía existe en las versiones actuales del kernel, su trabajo es como sigue:

1. Si en los parámetros de `clone()` existe la bandera para `vfork()`, es invocado.
2. En `copy_process()`, el miembro `vfork_done` dentro de `task_struct` es puesto en `NULL`.
3. Si `do_fork()` es exitoso, `vfork_done` apunta a una dirección específica.
4. Cuando el hijo corre, el padre espera a que el hijo de una señal con el puntero en `vfork_done`.
5. En `mm_release()`, que se invoca al finalizar un hilo, se revisa si `vfork_done` es `NULL`, si no lo está, el padre recibe la señal.
6. Se regresa a `do_fork()`, el padre se despierta.

Si todo va bien, el hijo se ejecuta en un espacio de direccionamiento nuevo, y el padre ejecuta en su espacio original.

El uso de `vfork()` es en escenarios de bifurcación donde se llama a `exec()` y se debe de tener un control si `exec()` falla.

Implementación de Hilos

Linux no tiene un concepto de hilos, se implementan todos los hilos como procesos estándar.

No se tiene un algoritmo para planificación de hilos especial ni estructuras de datos para representar hilos.

Un hilo es solo un proceso que comparte recursos (como el espacio de direcciones) con otros procesos de su misma jerarquía.

Cada hilo tiene propiedades en `task_struct` pero aparece como un proceso para el kernel.

Cada hilo tiene su `task_struct` cuyos miembros tienen los mismos apuntadores.

Hilos de Usuario

Los hilos se crean igual que las tareas normales (procesos), pero `clone()` recibe parámetros especiales correspondientes a qué recursos serán compartidos.

Las llamadas `fork()`, `vfork()` y el hilo, se verían de la siguiente manera:

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

// vfork()
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);

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

Las banderas recibidas por `clone()` están definidas en *include/uapi/linux/sched.h*:

- `CLONE_VM`: Padre e hijo comparten mismo espacio de direccionamiento.
- `CLONE_FS`: Padre e hijo comparten información del sistema de archivos.
- `CLONE_FILES`: Padre e hijo comparten lista de archivos abiertos.
- `CLONE_SIGHAND`: Padre e hijo comparten manejadores de señales y señales bloqueadas.
- `CLONE_PTRACE`: Permite que el hijo sea monitoreado con *ptrace*.
- `CLONE_VFORK`: El padre se duerme hasta que el hijo termine.
- `CLONE_PARENT`: El hijo tendrá el mismo padre que el padre.
- `CLONE_THREAD`: Padre e hijo forman parte del mismo grupo de procesos/hilos.
- `CLONE_NEWNS`: Se creará un nuevo espacio con nombre para el hijo.
- `CLONE_SYSVSEM`: Padre e hijo comparten la semántica *SEM_UNDO* de semáforos SysV.
- `CLONE_SETTLS`: Crea un *Thread Local Storage* (alojar variables globales) para el hijo.
- `CLONE_PARENT_SETTID`: Definir el *Task ID* en el padre.
- `CLONE_CHILD_CLEARTID`: Borrar el *Task ID* en el hijo.
- `CLONE_UNTRACED`: No permite activar `CLONE_PTRACE` en el hijo.
- `CLONE_CHILD_SETTID`: Definir el *Task ID* en el hijo.
- `CLONE_NEWUTS`: Nuevo espacio de nombre para información de *Unix Timesharing System*.
- `CLONE_NEWIPC`: Nuevo espacio de nombre para información de Comunicación entre procesos compatible SysV o POSIX.
- `CLONE_NEWUSER`: Nuevo espacio de nombre para *User/Group ID*.
- `CLONE_NEWPID`: Nuevo espacio de nombre para gestión de procesos en contenedores (LXC).
- `CLONE_NEWNET`: Espacio de nombre para gestión de conexiones de red.
- `CLONE_IO`: Crear política de planificación de I/O.

Hilos de Kernel

Los hilos de kernel son procesos especiales, que todo el tiempo, de principio a fin, se mantienen en ejecución en espacio de kernel.

Los hilos de kernel pueden ser creados o invocados por diferentes componentes del kernel, incluyendo controladores y módulos, pero por un hilo de kernel inicial *kthreadd*, que es la contraparte del Proceso 1.

Los hilos de kernel no cuentan con un espacio de direcciones, ya que no cuentan con protección de memoria (¡están en espacio de kernel!).

Los hilos de kernel no hacen cambio de contexto en ningún momento; pero son gestionados por el planificador de tareas, a la par que los procesos de usuario.

Para ver hilos de kernel, ejecutar `ps -ef`.

El encargado de crear hilos de kernel está en <*linux/kthread.h*>:

```
1 struct task_struct *kthread_create(int (*threadfn)                    ❶
2                                    (void *data),                      ❷
3                                    void *data,
4                                    const char namefmt[], ...);        ❸
```

❶ Qué función será ejecutada como hilo de kernel

❷ Qué parámetros recibirá dicha función que será ejecutada como hilo de kernel

❸ Cómo se identificará al hilo de kernel

El nuevo hilo de kernel es creado igual con `clone()` por *kthreadd*, pero no será ejecutado hasta que se invoque con `wake_up_process()` de manera explícita.

Si se quiere crear y ejecutar un hilo de kernel, se puede hacer con:

```
1 struct task_struct *kthread_run(threadfn, data, namefmt, ...)
```

Cuando es iniciado, se mantiene hasta que se llame `do_exit()` u otra rutina de kernel llame a `kthread_stop()`, recibiendo la dirección de la estructura `task_struct` recibida por `kthread_create()`.

```
1 int kthread_stop(struct task_struct *k)
```

Terminación de Procesos

Un proceso termina cuando invoca la llamada al sistema `exit()` de manera explícita o cuando regresa se su rutina principal.

También, puede terminarse de manera involuntaria, recibiendo una señal que no pueda ignorar o con una excepción.

La destrucción de un proceso es manejado por *kernel/exit.c*:

1. Invocar a `do_exit()`.
2. Definir la bandera `PF_EXITING` en `flags` dentro de `task_struct`.
3. Llamar a `del_timer_sync()` para eliminar cualquier temporizador de kernel.
4. Llamar a `exit_mm()` para liberar estructuras `mm_struct` que sean del proceso. Si no hay otro proceso que la comparta, es liberada.
5. Llamar a `exit_sem()` para liberar semáforos SysV/POSIX.
6. Llamar a `exit_files()` y `exit_fs()` para reducir el conteo de uso de objetos relacionados con descriptores de archivos y datos de sistema de archivos. Si un contador llega a 0, se destruye el descriptor.

7. Se define el código de salida del proceso, almacenado en `exit_code` dentro de `task_struct`.
8. Se llama a `exit_notify()` para enviar señales al padre, los hijos del que está siendo destruído son reasignados a otro padre dentro del grupo de hilos o al proceso *init*, y define el estado de salida, en `exit_state` dentro de `task_struct`, en `EXIT_ZOMBIE`.
9. Se llama a `schedule()` para cambiar a otro proceso. Como el proceso que está siendo destruído ya no está en el planificador.

Todos los objetos asociados al proceso (si era el único del grupo), son liberados. El proceso ya no puede ejecutarse, ni tiene espacio de direcciones), y está en `EXIT_ZOMBIE`.

Eliminación del Descriptor de Proceso

Aún existe su pila de kernel, la estructura `thread_info` y `task_struct`, para proporcionar información de su estado al padre. Una vez que el padre recibe la información, se liberan estas estructuras al sistema para su uso posterior.

Para la liberación de las últimas estructuras:

1. Se invoca a `release_task()`.
2. Llama a `__exit_signal()`, que llama a `__unhash_process()`, `que llama a `detach_pid()` para eliminar el proceso de la lista de tareas.
3. `__exit_signal()` libera recursos del proceso muerto y actualiza estadísticas.
4. Si el proceso era el último miembro de un grupo de hilos, y el líder es zombie, `release_task()` notifica al padre del líder zombie.
5. `release_task()` llama a `put_task_struct()` para liberar páginas que contengan la pila de kernel y `thread_info` y actualiza el cache *SLAB*.

Procesos huerfanos

Cuando el padre finaliza antes que el hijo, debe de existir un mecanismo para que los huerfanos sean asignados a un nuevo padre, ya que estos podrían volverse zombies, gastando memoria del sistema.

La reasignación puede activarse al recibir `exit()`, y los candidatos son dentro del grupo de hilos, procesos padres del padre, o el proceso *init*.

Dentro de `exit_notify()` se llama a `forget_original_parent()` que llama a `find_new_reaper()`.

```
static struct task_struct *find_new_reaper(struct task_struct *father,
                                           struct task_struct *child_reaper) {
    struct task_struct *thread, *reaper;

    thread = find_alive_thread(father);
    if(thread) return thread;
    if(father->signal->has_child_subreaper) {
        for(reaper = father; !same_thread_group(reaper, child_reaper); reaper = reaper-> ↩
            real_parent) {
            if(reaper == &init_task) break;
            if(!reaper->signal->is_child_subreaper) continue;
            thread = find_alive_thread(reaper);
            if(thread) return thread;
        }
    }
    return child_reaper;
}
```

Planificación de Procesos

El kernel usa *Completely Fair Scheduler* (CFS), que fue agregado en la versión 2.6.23.

Los procesos se dividen a grandes rasgos en 2 tipos de categorías: Procesos dependientes de I/O y procesos dependientes de CPU.

Los procesos dependientes de I/O son aquellos cuya mayor parte del tiempo esperan una respuesta a una perición de I/O. Estos procesos tienen tiempos de uso de procesador cortos, porque eventualmente se bloquean esperando por I/O.

I/O puede ser interrupción de teclado, de ratón, de red o de disco.

Los procesos dependientes de CPU son aquellos que dedican la mayor parte del tiempo a ejecución de código para CPU. Tienden a usar tiempo de procesador hasta que son detenidos por el planificador porque no se bloquean esperando peticiones de I/O a menudo.

Una política de ejecución justa puede ser que los procesos dependientes de CPU sean colocados en ejecución menos veces pero por más tiempo, pero que puedan ser bloqueados lo más pronto posible si se recibe una interrupción de I/O para tener mejor respuesta hacia el humano.

Existen procesos cuyo comportamiento puede ser aleatorio, que pueden estar bloqueados esperando interrupción de teclado pero en otro momento pueden estar haciendo un cálculo de CPU intensivo.

Una política de planificación debe de poder en lo mejor posible el objetivo de tener una capacidad de respuesta rápida (baja latencia) y a la vez una utilización máxima (alto rendimiento), sin conocer de antemano el comportamiento futuro de diferentes procesos.

Se le da preferencia siempre a los procesos dependientes de I/O que a los dependientes de CPU.

Prioridad de Procesos

Un algoritmo que planifica los procesos integrado en CFS es uno basado en prioridad, el objetivo es asignar los procesos en la cola de ejecución en base a su valor asignado en necesidad de tiempo de procesador.

Los procesos con prioridad mayor son ejecutados antes que los de prioridad menor. Los procesos con la misma prioridad son encolados en modo *round-robin*.

Linux proporciona 2 rangos para prioridad. El de herencia UNIX, con valores de -20 a 19, donde los negativos tienen mayor prioridad, siendo 0 por omisión.

La prioridad de procesos en ejecución puede verse con `ps -el`.

El otro rango de prioridades es el de *tiempo real*, con valores del 0 al 99, en donde los positivos mayores tienen manor prioridad. Los procesos que manejen prioridad de *tiempo real* tienen más peso que los procesos de política *nice* mencionados anteriormente.

Se pueden ver con `ps -eo state,uid,pid,ppid,rtprio,time,comm`.

Pedazos de Tiempo

Un pedazo de tiempo (*timeslice* o *quantum*) es un valor que representa cuanto tiempo debe de estar en CPU una tarea hasta que sea cambiada por otra. El planificador define un *timeslice* por defecto como constante, pero este valor cambia pasando el tiempo, en base a trabajo previo.

Un *quantum* muy largo ocaciona que el sistema tenga baja capacidad de respuesta interactiva; un *quantum* muy corto ocaciona que se desperdicie demasiado CPU para hacer los cambios de un proceso a otro.

El planificador de procesos también necesita de tiempo de ejecución en CPU para actualizar sus colas de ejecución, y su tiempo debe de ser corto.

CFS no asigna un valor duro de *timeslice* como en otros Sistemas Operativos, sino que asigna un *quantum* proporcional a uso de procesador. El tiempo de uso de procesador para un proceso es otorgado en base a la carga del sistema. La política *nice* se define como peso.

Para CFS, el siguiente proceso en turno se basa en cuanta proporción de procesador ha consumido anteriormente, si ha consumido una proporción menor que el proceso anterior, entra inmediatamente.

Política de Planificación

Con CFS, ocurre el siguiente escenario:

1. Se tiene un proceso dependiente de I/O y un proceso dependiente de CPU, luego entonces, se tienen 2 procesos en total
2. El CPU se divide proporcionalmente entre el total de procesos: 50%
3. El proceso I/O consume menos del 50% de CPU, por lo que el proceso CPU puede consumir más del 50% para evitar que el procesador esté ocioso
4. El proceso I/O despierta para atender una interrupción interactiva, como es el proceso que menos CPU ha consumido, tiene preferencia de ejecución por encima del otro proceso.
5. El proceso I/O vuelve a dormir, por lo que el proceso CPU tiene la libertad de seguir consumiendo procesador.

El Algoritmo de Planificación

Clases de Planificadores

En Linux el planificador es modular, permitiendo diferentes algoritmos para diferentes tipos de procesos, a estos algoritmos se les denomina clases.

Cada clase tiene una prioridad, la clase con prioridad más alta que tenga procesos en cola tiene turno.

Planificador completamente justo

CFS se basa en un concepto simple:

> Se planifican los procesos como si el sistema estuviera corriendo sobre un procesador multitareas perfectamente ideal.

En este sistema ideal, cada proceso recibiría $^{1}/_{n}$ del tiempo de procesador, donde *n* es el número de procesos en cola de ejecución, y se ejecutan todos los procesos de manera que todos los procesos en cola de ejecución han estado en CPU la misma cantidad de tiempo equitativamente.

Por ejemplo:

- Se tienen 2 procesos en cola de ejecución.
- Cada procesos es ejecutado durante 5 milisegundos.
- Cada proceso recibirá 100% de procesador.
- Idealmente, ambos procesos se ejecutarían *simultáneamente* durante 10 milisegundos, con el 50% de procesador.

Hay un costo al hacer cambios de contexto, volcados y restauración de estados, actualización de contadores, etcétera.

CFS ejecuta los procesos durante un periodo de tiempo, en modo *round-robin*, seleccionando el proceso que ha tenido menos tiempo de ejecución. En vez de asignar a cada proceso un *quantum* predefinido, CFS calcula cuanto tiempo debe de estar un proceso en ejecución en función al total de los procesos existentes.

En vez de usar un valor *nice* para calcular un *quantum*, lo usa como un peso para la proporción de procesador que puede recibir el proceso. Prioridades más bajas reciben un peso menor relativo al valor *nice* por defecto, y las prioridades más altas reciben un peso mayor.

Cada proceso se ejecuta por un *quantum* proporcional a su peso dividido por el total de los pesos de todas las tareas.

Para calcular el *quantum*, CFS calcula una aproximación de la duración de planificación más pequeña para la multitarea perfecta, o latencia objetivo.

Si la latencia objetivo es de 20 milisegundos y se tienen 2 procesos con la misma prioridad, cada uno será ejecutado por 10 milisegundos; si se tienen 4 procesos, serían 5 milisegundos, si son 20 procesos, sería 1 milisegundo.

Si el número de procesos tiende al infinito, la latencia ideal tiende a cero. CFS impone un piso de latencia, o granularidad mínima, que es de 1 milisegundo. Si el número de procesos tiende al infinito, cada uno será ejecutado al menos 1 milisegundo.

Si se tiene un proceso con peso 0 y otro proceso con peso 5, y la latencia calculada es de 20 milisegundos, el proceso peso 0 recibirá 15 milisegundos y el proceso peso 5 recibirá 5 milisegundos, así sea que un proceso tiene peso 10 y el otro tiene peso 15.

La proporción de tiempo de procesador que recibe cada proceso es determinada por la diferencia relativa de pesos *nice* de todos los procesos ejecutables.

CFS le asigna a cada proceso una proporción *justa* de tiempo de procesador.

La Implementación en Linux

La implementación de CFS se encuentra en *kernel/sched/fair.c*.

Todos los planificadores en cualquier SO y en Linux deben de tener una noción de cuanto tiempo ha estado en ejecución un proceso dado. Con cada *tick* del reloj del sistema, el *quantum* de un proceso en ejecución se decrementa, cuando llega a cero, el proceso es detenido para ser cambiado por otro proceso que tenga mayor *quantum*.

Estructura de Entidad de Planificador

Aunque CFS no tiene un *quantum* predefinido, se debe de tener noción de cuanto tiempo ha estado un proceso en ejecución, para asegurarse que cada proceso tiene un uso *justo* y equitativo de procesador. La estructura `sched_entity` se define en *<linux/sched.h>*:

```
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;

    u64 exec_start;
    u64 sum_exec_runtime;
    u64 vruntime;
    u64 prev_sum_exec_runtime;
    u64 nr_migrations;
};
```

Esta estructura también está apuntada en el descriptor de proceso (`task_struct`).

Virtual Runtime

La variable `vruntime` almacena el *tiempo de ejecución virtual* del proceso, que es el tiempo que ha estado en ejecución, medido en nanosegundos. Esta variable se usa para aproximarse al tiempo ideal para CFS. En el ambiente ideal, la variable `vruntime` tendría el mismo valor en todos los procesos existentes.

La función `update_curr()` definida en *kernel/sched/fair.c* se encarga de actualizar este valor:

```
static void update_curr(struct cfs_rq *cfs_rq) {
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if(unlikely(!curr)) return;

    delta_exec = now - curr->exec_start;
```

```
    if(unlikely((s64)delta_exec <= 0)) return;

    curr->exec_start = now;

    schedstat_set(curr->statistics.exec_max, max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq, exec_clock, delta_exec);

    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfq_rq);

    if(entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cpuacct_charge(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

`update_curr()` calcula el tiempo de ejecución del proceso en turno y guarda el valor en `delta_exec`, luego pesa el tiempo por el número de procesos, después se actualiza `vruntime` con el valor apropiado de peso.

`update_curr()` es llamado siempre por el temporizador del sistema y cuando un proceso entra en ejecución o se bloquea. En base a los cálculos hechos, CFS puede decidir el siguiente turno.

Selección del Proceso

La selección es simple, el siguiente proceso en turno es aquel cuyo valor de `vruntime` sea el menor.

CFS usa un *árbol rojinegro* para manejar la lista de procesos ejecutables y encontrar el proceso con el `vruntime` más chico.

Un árbol rojinegro, o *rbtree*, es un tipo de árbol binario de búsqueda auto balanceado, donde se almacenan nodos de datos arbitrarios, identificados por una clave, para una búsqueda más eficiente.

Se tiene este *rbtree* con todos los procesos ejecutables del sistema en donde la clave de cada nodo es el `vruntime` de los procesos.

El `vruntime` más bajo está más a la izquierda del árbol. Si se recorre el árbol de la raiz hacia el hijo de la izquierda hasta llegar a la hoja, se encontrará el proceso que se busca.

```
/* kernel/sched/fair.c */

static struct sched_entity *__pick_next_entity(struct sched_entity *se) {
    struct rb_node *next = rb_next(&se->run_node);

    if (!next)
        return NULL;

    return rb_entry(next, struct sched_entity, run_node);
}

/* ... */

static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity * ↩
    curr) {
    struct sched_entity *left = __pick_first_entity(cfs_rq);
    struct sched_entity *se;
```

```

    if(!left || (curr && entity_before(curr, left)))
        left = curr;

    se = left;

    if(cfs_rq->skip == se) {
        struct sched_entity *second;

        if (se == curr) {
            second = __pick_first_entity(cfs_rq);
        } else {
            second = __pick_next_entity(se);
            if (!second || (curr && entity_before(curr, second)))
                second = curr;
        }

        if (second && wakeup_preempt_entity(second, left) < 1)
            se = second;
    }

    if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
        se = cfs_rq->last;

    if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
        se = cfs_rq->next;

    clear_buddies(cfs_rq, se);

    return se;
}
```

Si no hay ningún proceso ejecutable, se entra en la tarea ociosa (*idle*).

Procesos en el árbol

CFS se encarga de agregar procesos a *rbtree* de ejecución. Los procesos que se agregan son aquellos que despiertan o que son creados recientemente.

```
/* kernel/sched/fair.c */

static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;

    while(*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }

    if(leftmost)
        cfs_rq->rb_leftmost = &se->run_node;

```

```
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}

/* ... */

static void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags) {
    if(!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
        se->vruntime += cfs_rq->min_vruntime;

    update_curr(cfs_rq);
    enqueue_entity_load_avg(cfs_rq, se);
    account_entity_enqueue(cfs_rq, se);
    update_cfs_shares(cfs_rq);

    if(flags & ENQUEUE_WAKEUP) {
        place_entity(cfs_rq, se, 0);
        enqueue_sleeper(cfs_rq, se);
    }

    update_stats_enqueue(cfs_rq, se);
    check_spread(cfs_rq, se);
    if(se != cfs_rq->curr)
        __enqueue_entity(cfs_rq, se);
    se->on_rq = 1;

    if (cfs_rq->nr_running == 1) {
        list_add_leaf_cfs_rq(cfs_rq);
        check_enqueue_throttle(cfs_rq);
    }
}
```

La función `__enqueue_entity()` tiene un `while()`, que atraviesa el árbol para buscar la clave, se mueve al hijo de la izquierda o derecha y almacena en cache cuál es el que se encuentra más a la izquierda.

En `rb_link_node()` se inserta el proceso en la rama, y en `rb_insert_color()` se actualiza la propiedad de balanceo del árbol.

Eliminar procesos del árbol

CFS elimina procesos del árbol cuando un proceso se bloquea o termina.

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
    if(cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}

/* ... */

static void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags) {
    update_curr(cfs_rq);
    dequeue_entity_load_avg(cfs_rq, se);

    update_stats_dequeue(cfs_rq, se);

```

```
    clear_buddies(cfs_rq, se);

    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);

    if (!(flags & DEQUEUE_SLEEP))
        se->vruntime -= cfs_rq->min_vruntime;

    return_cfs_rq_runtime(cfs_rq);

    update_min_vruntime(cfs_rq);
    update_cfs_shares(cfs_rq);
}
```

Eliminar un proceso del árbol es más simple porque se cuenta con un `rb_erase()` que se encarga del trabajo.

Punto de Entrada del Planificador

El principal punto de entrada del planificador es la función y llamada al sistema `schedule()`, definida en *kernel/sched/core.c*. Esta función es usada por el resto del kernel para invocar al planificador de procesos, decidiendo qué proceso colocar en el CPU.

`schedule()` es genérica de las clases del planificador, entre sus tareas, está `pick_next_task()`, que recorre las clases iniciando con la de mayor prioridad.

```
static inline struct task_struct *pick_next_task(struct rq *rq, struct task_struct *prev) {
    const struct sched_class *class = &fair_sched_class;
    struct task_struct *p;

    if(likely(prev->sched_class == class && rq->nr_running == rq->cfs.h_nr_running)) {
        p = fair_sched_class.pick_next_task(rq, prev);
        if(unlikely(p == RETRY_TASK))
            goto again;

        if(unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev);

        return p;
    }

again:
    for_each_class(class) {
        p = class->pick_next_task(rq, prev);
        if(p) {
            if(unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    BUG(); /* the idle class will always have a runnable task */
}
```

`likely()` y `unlikely()` son macros que agrupan directivas de compilador para optimización de condiciones. No se recomienda usarlas a la ligera.

Como CFS es la clase más habitual para procesos normales, que son la mayoría, se selecciona el siguiente proceso si el número de procesos ejecutables es igual al número de procesos en la clase CFS. Cada clase tiene su propia implementación de `pick_next_task()`, que regresa un puntero a su siguiente proceso ejecutable o `NULL`.

La primera clase que regrese un puntero es la que tiene preferencia de ejecución.

Dormir proceos

Los procesos dormidos o bloqueados, están en un estado que no es de ejecución. El planificador no puede seleccionar tareas que no quieren ejecutarse.

Una tarea debe de dormir por cualquier razón, principalmente esperando algún evento. El evento puede ser una cantidad de tiempo (temporizador), esperando datos de I/O u otro evento de hardware.

Una tarea puede dormir de manera involuntaria su trata de obtener un semáforo en el kernel, la razón más común es espera de I/O, como `read()` o interrupción de teclado.

El kernel coloca la tarea bloqueada en una cola de espera, la elimina del *rbtree* de ejecución y se llama a `schedule()` para seleccionar el siguiente proceso en turno.

Cuando despiertan, la tarea se marca como ejecutable, se elimina de la cola de espera y se agrega al *rbtree* de ejecución.

La cola de espera es una lista de procesos esperando a un evento.

Las colas de espera son definidas por `wait_queue_head_t`, creadas estáticamente por `DECLARE_WAITQUEUE()` o dinámicamente por `init_waitqueue_head()`.

Los mismos procesos se colocan en la cola de espera y se bloquean. Cuando un evento asociado con una cola de espera ocurre, los procesos en la cola son despertados.

Una condición de carrera común es mandarse a dormir e inmediatamente el evento ocurre, dejando el proceso dormido indefinidamente.

```
1  DEFINE_WAIT(wait);                                          ❶
2
3  add_wait_queue(q, &wait);                                   ❷
4  while(!condicion) {
5      prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);         ❸
6      if(signal_pending(current))
7          /* ... */
8          break;                                              ❹
9      schedule();
10 }
11 finish_wait(&q, &wait);                                     ❺
```

❶ Se crea una entrada en la cola de espera con la macro `DEFINE_WAIT()`

❷ Se agrega a la cola de espera con `add_wait_queue()`. El proceso se despierta cuando la condición ocurre usando `wake_up()`

❸ Se llama a `prepare_to_wait()` para cambiar de estado, también se regresa la tarea a la cola de espera si es necesario. una señal despierta al proceso, por lo que se revisa la señal pendiente

❹ Si la tarea despierta, revisa su condición, si es lo esperado, se sale del bucle, sino, se llama a `schedule()`

❺ Fuera del bucle, se cambia al estado `TASK_RUNNING` y se elimina de la cola de espera

Si la condición ocurre antes de dormir, el bucle termina y la tarea no se bloquea por error.

Antes de llamar a `schedule()` es conveniente liberar bloqueas y volverlos a colocar al despertar.

Ejemplo: *fs/notify/inotify/inotify_user.c*:

```
1  static ssize_t inotify_read(struct file *file,
2                              char __user *buf,
3                              size_t count,
4                              loff_t *pos) {
5      struct fsnotify_group *group;
```

```
    struct fsnotify_event *kevent;
    char __user *start;
    int ret;
    DEFINE_WAIT_FUNC(wait, woken_wake_function);

    start = buf;
    group = file->private_data;

    add_wait_queue(&group->notification_waitq, &wait);
    while(1) {
        mutex_lock(&group->notification_mutex);
        kevent = get_one_event(group, count);
        mutex_unlock(&group->notification_mutex);

        pr_debug("%s: group=%p kevent=%p\n", __func__, group, kevent);

        if(kevent) {
            ret = PTR_ERR(kevent);
            if(IS_ERR(kevent))
                break;
            ret = copy_event_to_user(group, kevent, buf);
            fsnotify_destroy_event(group, kevent);
            if(ret < 0)
                break;
            buf += ret;
            count -= ret;
            continue;
        }

        ret = -EAGAIN;
        if(file->f_flags & O_NONBLOCK)
            break;
        ret = -ERESTARTSYS;
        if(signal_pending(current))
            break;

        if(start != buf)
            break;
        wait_woken(&wait, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
    }
    remove_wait_queue(&group->notification_waitq, &wait);

    if(start != buf && ret != -EFAULT)
        ret = buf - start;
    return ret;
}
```

Despertar procesos

Para despertar el proceso es a través de `wake_up()`, el cual despierta todos los procesos que se encuentren en una cola de espera dada. Llama a `try_to_wake_up()`, que les cambia al estado `TASK_RUNNING`, llama a `enqueue_task()` para agregarlas al *rbtree* de ejecución, y define la bandera `need_resched` si la prioridad de la tarea recién despierta es mayor que la tarea en ejecución.

Un ejemplo común, es cuando VFS llama a `wake_up()` para despertar a todos los procesos que esperan datos del Sistema de Archivos.

El que una tarea sea despertada no significa que el evento que espera la tarea ha ocurrido, es conveniente que pase a dormir dentro de un bucle para asegurarse que la condición que se está esperando de verdad haya ocurrido.

Cambios de Contexto

Un cambio de contexto es la acción de cambiar una tarea en ejecución por otra, la acción es por `context_switch()`, definida en *kernel/sched/core.c*. Esta función es llamada por `schedule()`, con las siguientes actividades:

- Llama a `switch_mm()`, declarada en <*asm/mmu_context.h*> (*arch/arm/include/asm/mmu_context.h*), que cambia el mapeo de memoria virtual del proceso anterior al nuevo proceso.
- Llama a `switch_to()` declarada en <*asm/switch_to.h*> que cambia el estado del procesador del proceso previo al nuevo. Almacena y restaura la información de pila y los registros de CPU por cada proceso.

Si el kernel llamara a `schedule()` cuando el código lo llama de manera explícita, los procesos de usuario correrían indefinidamente. La bandera `need_resched` se activa si una replanificación es necesaria. La bandera es ajustada por `scheduler_tick()` cuando un proceso debe de ser pausado y por `try_to_wake_up()` cuando un proceso tiene prioridad mayor que el que está en ejecución debe de ser despertado.

El kernel revisa la bandera, y llama a `schedule()` para cambiar a un nuevo proceso. La bandera le avisa al kernel para invocar el planificador lo más pronto posible.

Table 1: Funciones que manipulan `need_resched`

Función	**Propósito**
`set_tsk_need_resched()`	Define `need_resched` en un proceso dado
`clear_tsk_need_resched()`	Desactiva `need_resched` en un proceso dado
`need_resched()`	Verifica si la bandera está activa

Al regresar a espacio de usuario o de una interrupción, se revisa la bandera `need_resched`, si lo está, se llama al planificador.

La bandera es por proceso, definida en `thread_info`, porque es más rápido acceder al valor en el descriptor de proceso que en una variable global.

La *preempción* ocurre cuando el kernel regresa a espacio de usuario, la bandera `need_resched` está activa, y se llama al planificador.

Si es seguro regresar a espacio de usuario de la tarea actual, es seguro elegir otra a ejecutar, después de una interrupción o después de ejecutar una llamada al sistema.

El regresar de espacio de kernel a espacio de usuario es dependiente de arquitectura, y está descrito en *arch/arm/nwfpe/entry.S* en ensamblador.

El mismo kernel de Linux puede llamarse a congelar (*preempt*) a sí mismo. El kernel puede dormir una tarea en espacio de kernel mientras no tenga un bloqueo explícito.

En `thread_info` se tiene un miembro `preempt_count`, que empieza en 0 y se incrementa por cada bloqueo que es adquirido y se decrementa por cada bloqueo liberado, cuando está en 0, el kernel puede hacer *preemption* a sí mismo.

Al regresar de una interrupción, y se está aún en espacio de kernel, se revisa a `need_resched` y a `preempt_count`, si ambos están en 0, otras tareas más importantes pueden entrar en ejecución.

Si `preempt_count` es mayor a 0, no se invoca al planificador, y la interrupción regresa a la misma tarea que está en turno.

El kernel puede hacerse *preempt* explícitamente, cuando una tarea en el kernel se bloquea o llama explícitamente a `schedule()`.

El código que llame explícitamente a `schedule()` debe de estar conciente que es seguro para llamar al planificador.

El kernel se duerme a sí mismo cuando:

- Cuando termina un manejador de interrupción, antes de regresar a espacio de kernel

- Cuando el código de kernel se vuelve *preemptible*
- Si una tarea en el kernel llama a `schedule()` explícitamente
- Si una tarea en el kernel se bloquea

Políticas de Tiempo Real

Linux proporciona dos políticas de planificación para tiempo real, `SCHED_FIFO` y `SCHED_RR`, que no son manejadas por CFS, sino por *kernel/sched/rt.c*.

`SCHED_FIFO` implementa Primero en Entrar, Primero en Salir sin *quantum*, tiene mayor preferencia que las tareas de CFS. Estas tareas se ejecutan hasta que se bloqueen o que indiquen que se duermen explícitamente, o pueden ejecutarse indefinidamente. Solo otra tarea `SCHED_FIFO` de mayor prioridad o de `SCHED_RR` puede tomar turno siguiente. 2 procesos `SCHED_FIFO` de la misma prioridad corren en *round-robin*.

`SCHED_RR` ejecuta cada proceso hasta que termine un *quantum* predeterminado, en *round-robin*. El siguiente turno es para otra `SCHED_RR` de mayor prioridad, las de menor prioridad no pueden seguir, aunque el *quantum* haya terminado.

Ambas políticas tienen prioridades estáticas, del 0 al 99.

Llamadas al sistema del Planificador

Llamada al Sistema	**Descripción**
`nice()`	Define un peso de prioridad en CFS
`sched_setscheduler()`	Define una política de planificación al proceso
`sched_getscheduler()`	Recibe qué política de planificación tiene el proceso
`sched_setparam()`	Define una prioridad de tiempo real
`sched_getparam()`	Recibe qué prioridad de tiempo real tiene el proceso
`sched_get_priority_max()`	Obtiene la prioridad de tiempo real máxima
`sched_get_priority_min()`	Obtiene la prioridad de tiempo real mínima
`sched_rr_get_interval()`	Obtiene el tamaño del *quantum* de `SCHED_RR`
`sched_setaffinity()`	Define la afinidad de procesador del proceso
`sched_getaffinity()`	Recibe la afinidad de procesador del proceso
`sched_yield()`	Salirse del procesador temporalmente