

PROCESSI

Il concetto di processo è fondamentale: di solito è definito come una istanza di un programma in esecuzione. Perciò se 16 utenti stanno usando vi contemporaneamente, sono presenti 16 processi separati anche se essi possono condividere lo stesso codice eseguibile. I processi sono anche definiti *task* o *thread* nel codice sorgente di Linux.

Processi, processi “leggeri” e thread

Il termine processo è spesso usato con diversi significati. Qui viene adottata la definizione classica dei testi sui sistemi operativi: processo è una istanza di un programma in esecuzione. Lo si può vedere come una collezione di strutture dati che descrivono quanto è avanzata l'esecuzione del programma.

I processi sono come gli esseri umani; nascono, conducono una vita più o meno significativa, possono generare uno o più processi figli ed eventualmente muoiono. Una differenza è che il sesso non è veramente molto comune tra i processi – ognuno ha un solo genitore. Dal punto di vista del kernel, scopo di un processo è di agire come un'entità a cui sono allocate risorse di sistema (CPU, memoria, ecc.).

Quando un processo viene creato, è praticamente identico al genitore. Riceve una copia logica dello spazio di indirizzamento ed esegue lo stesso codice del padre, a partire dall'istruzione seguente alla chiamata di sistema che lo ha creato. Anche se padre e figlio condividono le pagine del codice (text), hanno copie separate dei dati (stack e heap), così che il riferimento del figlio ad una locazione di memoria è invisibile per il padre e viceversa.

Mentre le prime versioni di Unix adottavano questo semplice modello, le più recenti supportano *applicazioni multithread*, programmi utente che hanno molti flussi di esecuzione relativamente indipendenti che condividono una gran parte delle strutture dati. Un processo risulta quindi composto da vari *user thread* o semplicemente *thread*, ognuno dei quali rappresenta un flusso di esecuzione. La maggior parte delle applicazioni multithread è scritta usando funzioni di librerie standard chiamate *pthread* (POSIX *thread*).

Vecchie versioni del kernel di Linux non offrivano alcun supporto al multithread. Dal punto di vista del kernel, erano applicazioni normali. Il flusso multiplo di esecuzione veniva creato, gestito e schedulato interamente in modalità utente per mezzo delle librerie pthread.

Una tale implementazione non era troppo soddisfacente. Ad esempio, si ipotizzi che un programma di scacchi usi due thread: uno controlla la scacchiera grafica, attende le mosse del giocatore umano e mostra quelle del computer, mentre l'altro considera la mossa successiva. Mentre il primo attende la mossa del giocatore, il secondo dovrebbe elaborare in continuo sfruttando il tempo morto. Comunque se si tratta di un processo singolo, il primo thread non può invocare una chiamata di sistema bloccante per attendere l'azione del giocatore, altrimenti sarebbe bloccato anche il secondo thread. Deve invece applicare tecniche sofisticate non bloccanti per assicurare che il processo rimanga in attività.

Linux usa i *lightweight processes* (LWP) per offrire migliore supporto al multithread. Due LWP possono condividere alcune risorse, come spazio di indirizzamento o file aperti. Quando uno di essi modifica una risorsa condivisa, l'altro vede immediatamente il cambiamento. Naturalmente i due processi devono sincronizzarsi quando accedono alle risorse condivise.

Un modo di implementare applicazioni multithread consiste nell'associare un LWP a ogni thread. In questo modo i thread possono accedere allo stesso insieme di strutture dati semplicemente condividendo lo stesso

spazio di indirizzamento, lo stesso set di file aperti e così via. Allo stesso modo ogni thread può essere schedulato indipendentemente dal kernel, in modo che mentre uno è inattivo l'altro rimane in esecuzione. Esempi di librerie pthread POSIX sono Linux threads, Native POSIX Thread Library, e Generation Posix Threading Package.

Le applicazioni multithread meglio supportate sono quelle che usano i gruppi di thread. In Linux un gruppo di thread è un set di LWP che implementano una applicazione multithread e agiscono come un processo unico nei confronti delle chiamate di sistema tipo getpid(), kill(), _exit()).

Descrittore di processo

Il kernel deve avere una chiara visione di ciò che fa ogni processo. Deve conoscerne la priorità, se è in esecuzione su una CPU o se è bloccato su un evento, quale spazio di indirizzamento gli è stato assegnato, a quali file può accedere, e così via. Questo è il ruolo del *descrittore di processo* (PD), una struttura di tipo task_struct (equivalente al tipo task_t) i cui campi contengono tutte le informazioni relative ad un singolo processo. Per questo risulta piuttosto complesso: oltre ai numerosi campi che contengono gli attributi, contiene anche vari puntatori ad altre strutture che, a loro volta, contengono puntatori ad altre strutture.

```
/include/linux/sched.h
```

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;    /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth;        /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
    struct sched_info sched_info;
#endif

    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
    */
};
```

```

    */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
    /* ??? */
    unsigned long personality;
    unsigned did_exec:1;
pid_t pid;
pid_t tgid;
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively. (p->father can be replaced with
     * p->parent->pid)
     */
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent; /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

    /* PID/PID hash table linkage. */
struct pid pids[PIDTYPE_MAX];

    struct completion *vfork_done; /* for vfork() */
    int __user *set_child_tid; /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */

    unsigned long rt_priority;
    unsigned long it_real_value, it_real_incr;
    cputime_t it_virt_value, it_virt_incr;
    cputime_t it_prof_value, it_prof_incr;
    struct timer_list real_timer;
    cputime_t utime, stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
    unsigned long minflt, majflt;
/* process credentials */
    uid_t uid, euid, suid, fsuid;
    gid_t gid, egid, sgid, fsgid;
    struct group_info *group_info;

```

```

    kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
    struct user_struct *user;
#ifdef CONFIG_KEYS
    struct key *session_keyring; /* keyring inherited over fork */
    struct key *process_keyring; /* keyring private to this process (CLONE_THREAD) */
    struct key *thread_keyring; /* keyring private to this thread */
#endif
    int oomkilladj; /* OOM kill score adjustment (bit shift). */
    char comm[TASK_COMM_LEN];
/* file system info */
    int link_count, total_link_count;
/* ipc stuff */
    struct sysv_sem sysvsem;
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespace */
    struct namespace *namespace;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;

    void *security;
    struct audit_context *audit_context;

/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
    spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
    spinlock_t proc_lock;
/* context-switch lock */
    spinlock_t switch_lock;

/* journalling filesystem info */
    void *journal_info;

/* VM state */

```

```

struct reclaim_state *reclaim_state;

struct dentry *proc_dentry;
struct backing_dev_info *backing_dev_info;

struct io_context *io_context;

unsigned long ptrace_message;
siginfo_t *last_siginfo; /* For ptrace use. */
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
    wait_queue_t *io_wait;
/* i/o counters(bytes read/written, #syscalls */
    u64 rchar, wchar, syscr, syscw;
#ifdef CONFIG_BSD_PROCESS_ACCT)
    u64 acct_rss_mem1; /* accumulated rss usage */
    u64 acct_vm_mem1; /* accumulated virtual memory usage */
    clock_t acct_stimexpd; /* clock_t-converted stime since last update */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next;
#endif
};

```

Stato del processo

Come indica il nome, il campo state descrive cosa sta accadendo al processo. E' formato da un array di flag, ognuno dei quali descrive un possibile stato. Nella versione corrente di Linux gli stati sono mutualmente esclusivi: solo un flag è attivato, gli altri sono azzerati. Gli stati possibili sono i seguenti:

- **TASK_RUNNING**: il processo è in esecuzione o in attesa di essere eseguito
- **TASK_INTERRUPTIBLE**: il processo è sospeso (sleeping) fino a che non si verifica una data condizione: la ricezione di un interrupt hardware, il rilascio di una risorsa che il processo sta aspettando, l'invio di un segnale sono esempi di condizioni che possono risvegliare un processo.
- **TASK_UNINTERRUPTIBLE**: simile allo stato precedente, con la differenza che un segnale non può risvegliare il processo. Questo stato viene usato raramente. E' impiegato in certe condizioni quando un processo deve aspettare il verificarsi di un evento senza essere interrotto. Si usa quando un processo apre un file di dispositivo e il driver corrispondente inizia a interrogare il dispositivo. Il driver non deve essere interrotto finché il test non è completo altrimenti il dispositivo può rimanere in uno stato non prevedibile.
- **TASK_STOPPED**: l'esecuzione del processo è stata arrestata dopo aver ricevuto un segnale SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU.
- **TASK_TRACED**: l'esecuzione è stata arrestata da un debugger. Quando un processo è monitorato da

un altro (ad es. un debugger esegue una chiamata di sistema `ptrace()` su un programma test) ogni segnale può porre il processo in uno stato `TASK_TRACED`.

Due stati aggiuntivi possono essere memorizzati sia nel campo `state` che nel campo `exit_state` del PD; il processo raggiunge questi due stati solo quando viene terminato:

- `EXIT_ZOMBIE`: l'esecuzione del processo è terminata ma il genitore non ha ancora eseguito una chiamata di sistema `wait4()` o `waitpid()` per ottenere informazioni sul processo estinto. Prima che sia avvenuta la chiamata alla `wait()`, il kernel non può cancellare i dati contenuti nel PD del processo estinto, poiché il genitore potrebbe averne bisogno.
- `EXIT_DEAD`: è lo stato finale; il processo sta per essere rimosso dal sistema perché il genitore ha eseguito la chiamata di sistema `wait4()` o `waitpid()`. Cambiare lo stato da `EXIT_ZOMBIE` a `EXIT_DEAD` evita condizioni critiche (race conditions) dovute ad altri thread che eseguono chiamate `wait()` sullo stesso processo.

Il valore del campo `state` è impostato con un semplice assegnamento, ad es.

```
p->state = TASK_RUNNING;
```

Il kernel usa anche le macro `set_task_state` e `set_current_state`: esse impostano il valore di `state` per il processo indicato e per quello corrente, rispettivamente. Esse poi assicurano che l'assegnamento non venga mescolato ad altre istruzioni dal compilatore o dalla unità di controllo della CPU. Mescolare l'ordine delle istruzioni può portare ad esiti catastrofici.

Identificazione di un processo

In generale, ogni contesto di esecuzione che può venire schedato indipendentemente, deve avere il suo PD, perciò anche i LWP hanno le proprie strutture `task_struct`.

La corrispondenza uno a uno tra processo e PD rende l'indirizzo a 32 bit della struttura `task_struct` un mezzo utile al kernel per identificare il processo. Questi indirizzi sono chiamati puntatori ai PD, e sono il mezzo più usato dal kernel per referenziare i processi.

D'altra parte i sistemi Unix consentono agli utenti di identificare i processi per mezzo di un numero chiamato *ID di processo* (PID), che viene memorizzato nel campo `pid` del PD. I PID sono numerati in sequenza, e il PID dell'ultimo processo creato corrisponde al PID del precedente + 1. Chiaramente esiste un limite superiore ai valori di PID, raggiunto il quale il kernel deve ricominciare riciclando il più basso non in uso. Di default il massimo valore è 32.767 (`PID_MAX_DEFAULT - 1`); l'amministratore di sistema può ridurre questo limite scrivendo un valore inferiore in `/proc/sys/kernel/pid_max`. Nei sistemi a 64 bit l'amministratore può innalzare il limite fino a 4.194.303.

Quando ricicla i PID, il kernel deve gestire una bitmap (mappa di bit) `pidmap_array`, che indica quali PID sono in uso e quali sono disponibili. Poiché un frame di pagina contiene 32.768 bit, nei sistemi a 32 bit la `pidmap_array` è contenuta in una singola pagina. Nei sistemi a 64 bit alla bitmap possono essere aggiunte pagine se il kernel supera il limite; queste pagine non vengono mai rilasciate.

Linux associa un PID ad ogni processo o LWP, con una piccola eccezione per i sistemi multiprocessore. Questo sistema garantisce la massima flessibilità, perché ogni contesto di esecuzione può essere identificato.

D'altra parte i programmatori si aspettano che i thread dello stesso gruppo abbiano un PID comune. Per esempio deve essere possibile inviare un segnale indicando un PID che riguarda tutti i thread del gruppo.

Infatti lo standard POSIX 1003.1c impone che tutti i thread di una applicazione abbiano lo stesso PID.

Per adeguarsi a questo standard, Linux usa i gruppi di thread. L'identificatore comune è il PID del leader del gruppo, cioè del primo LWP del gruppo; il valore è memorizzato nel campo `tgid` del PD. La chiamata di sistema `getpid()` restituisce il valore di `tgid` del processo corrente al posto di `pid`, in modo che tutti thread della stessa applicazione condividano l'identificatore. La maggior parte dei processi appartengono ad un gruppo di thread con un solo membro; in quanto leader del gruppo, hanno `tgid` uguale a `pid`, per cui la `getpid()` funziona come previsto.

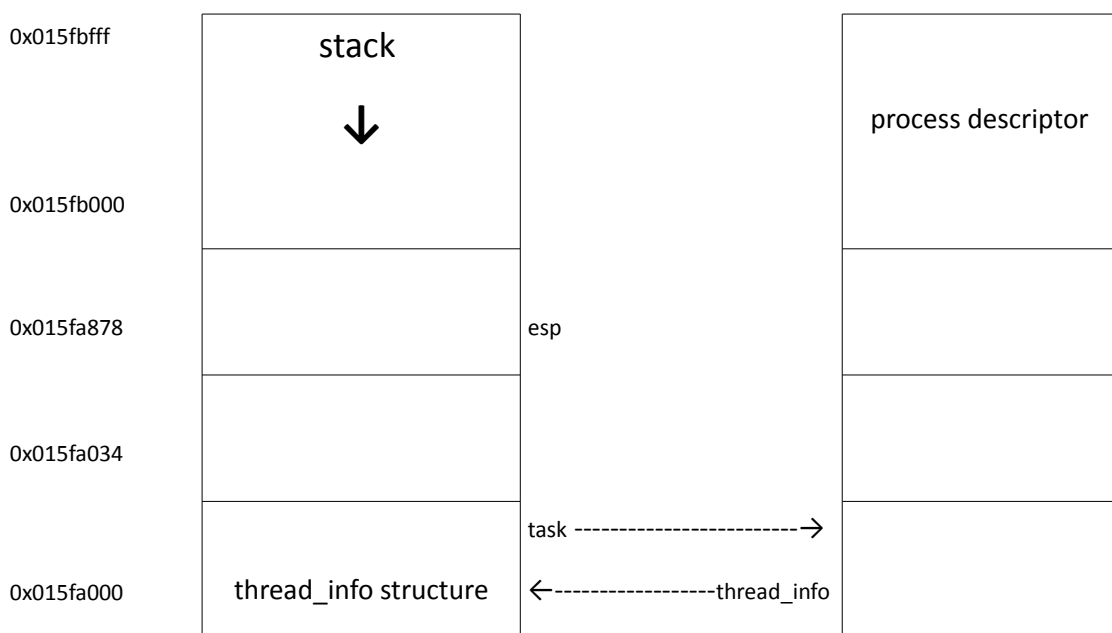
Si vedrà poi come ottenere il puntatore al PD a partire dal PID in modo efficiente. L'efficienza è importante perché molte chiamate di sistema come `kill()` usano il PID per indicare il processo interessato.

Gestione dei descrittori di processo

I processi sono entità dinamiche con una vita che può durare da pochi millisecondi a mesi. Perciò il kernel deve poter gestire molti processi contemporaneamente, e i PD sono mantenuti nella memoria dinamica piuttosto che nell'area assegnata in permanenza al kernel. Per ogni processo Linux immagazzina due differenti strutture dati in un'area di memoria unica per ogni processo: una piccola struttura collegata al PD, la struttura `thread_info`, e lo stack del processo in modalità del kernel. La dimensione di questa area è di 8192 bytes (2 frame di pagina). Per ragioni di efficienza il kernel memorizza quest'area in due frame consecutivi, con il primo allineato a un multiplo di 2^{13} ; questo può creare problemi quando è disponibile poca memoria, poiché la memoria libera può essere molto frammentata. Per questo il kernel può essere compilato in modo da includere le due strutture in un unico frame di pagina (4096 byte).

Un processo in modalità del kernel utilizza uno stack nel segmento dati del kernel, diverso da quello utilizzato in modalità utente. Poiché i kernel control paths (KCP) fanno poco uso dello stack, richiedono solo poche centinaia di byte di spazio. Perciò 8 KB è uno spazio ampio per stack e struttura `thread_info`. Comunque, quando stack e `thread_info` sono contenuti in un unico frame, il kernel usa stack aggiuntivi per evitare overflow provocati da interrupt ed eccezioni profondamente nidificati.

La struttura `thread_info` risiede all'inizio dell'area di memoria, e lo stack cresce verso il basso a partire dalla parte alta del frame. Inoltre le strutture `thread_info` e `task_struct` sono legate a vicenda per mezzo dei campi `task` e `thread_info`.



Il registro esp è il puntatore alla cima dello stack. Nei sistemi 80x86 lo stack inizia alla fine dell'area di memoria e cresce verso l'inizio. Dopo essere passato in modalità del kernel, lo stack del kernel di un processo è sempre vuoto, e perciò esp punta al byte immediatamente successivo allo stack.

Il valore di esp decresce man mano che i dati sono inseriti nello stack. Poiché la struttura thread_info occupa 52 byte, lo stack può espandersi fino a 8.140 byte.

Il linguaggio C rappresenta questa condizione con una union (/include/asm-i386/thread_info.h):

```

union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048]; /* 1024 per stack di 4 KB */
};

struct thread_info {
    struct task_struct *task; /* main task structure */
    struct exec_domain *exec_domain; /* execution domain */
    unsigned long flags; /* low level flags */
    unsigned long status; /* thread-synchronous flags */
    __u32 cpu; /* current CPU */
    __s32 preempt_count; /* 0 => preemptable, <0 => BUG */

    mm_segment_t addr_limit; /* thread address space:
                               0-0xBFFFFFFF for user-thread
                               0-0xFFFFFFFF for kernel-thread
                               */
    struct restart_block restart_block;
    unsigned long previous_esp; /* ESP of the previous stack in case
                                of nested (IRQ) stacks
                                */
    __u8 supervisor_stack[0];
};

/*
 * thread information flags
 * - these are process state flags that various assembly files may need to access
 * - pending work-to-be-done flags are in LSW
 * - other flags in MSW
 */
#define TIF_SYSCALL_TRACE 0 /* syscall trace active */
#define TIF_NOTIFY_RESUME 1 /* resumption notification requested */
#define TIF_SIGPENDING 2 /* signal pending */
#define TIF_NEED_RESCHED 3 /* rescheduling necessary */
#define TIF_SINGLESTEP 4 /* restore singlestep on return to user mode */
#define TIF_IRET 5 /* return with iret */
#define TIF_SYSCALL_AUDIT 7 /* syscall auditing active */
#define TIF_POLLING_NRFLAG 16 /* true if poll_idle() is polling TIF_NEED_RESCHED */
#define TIF_MEMDIE 17

```

```

#define _TIF_SYSCALL_TRACE (1<<TIF_SYSCALL_TRACE)
#define _TIF_NOTIFY_RESUME (1<<TIF_NOTIFY_RESUME)
#define _TIF_SIGPENDING (1<<TIF_SIGPENDING)
#define _TIF_NEED_RESCHED (1<<TIF_NEED_RESCHED)
#define _TIF_SINGLESTEP (1<<TIF_SINGLESTEP)
#define _TIF_IRET (1<<TIF_IRET)
#define _TIF_SYSCALL_AUDIT (1<<TIF_SYSCALL_AUDIT)
#define _TIF_POLLING_NRFLAG (1<<TIF_POLLING_NRFLAG)

```

Il kernel usa le macro `alloc_thread_info` e `free_thread_info` per allocare e rilasciare l'area di memoria che contiene lo stack e la struttura `thread_info`.

Identificazione del processo corrente

La stretta associazione tra stack in modalità del kernel e struttura `thread_info` appena descritta offre un beneficio in termini di efficienza: il kernel può ottenere facilmente l'indirizzo della struttura `thread_info` a partire dal valore del registro `esp`. Infatti se la union `thread_union` è lunga 8 KB (2^{13} byte), il kernel maschera i 13 bit meno significativi di `esp` per ottenere l'indirizzo base di `thread_info`; se la union è lunga 4 KB, il kernel maschera i 12 bit meno significativi.

L'operazione viene compiuta dalla funzione `current_thread_info()`:

```

static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__ ("andl %%esp,%0; \": "=r" (ti) : "0" (~(THREAD_SIZE - 1)));
    return ti;
}

```

che produce un codice assembly simile al seguente:

```

movl $0xffffe000, %ecx      /* or 0xfffff000 if 4 KB stack */
andl %esp, %ecx
movl %ecx, p

```

`p` quindi contiene il puntatore alla struttura `thread_info` del processo corrente.

Spesso il kernel ha bisogno invece dell'indirizzo del PD corrente; per ottenerlo usa la macro `current`:

```

static inline struct task_struct * get_current(void)
{
    return current_thread_info()->task;
}
#define current get_current()

```

Essa produce un codice assembly simile al seguente:

```

movl $0xffffe000, %ecx      /* or 0xfffff000 if 4 KB stack */
andl %esp, %ecx
movl (%ecx), p

```

Poiché il campo `task` è all'offset 0 nella struttura `thread_info`, `p` contiene ora il puntatore al PD del processo corrente. La macro `current` appare spesso nel codice del kernel come un prefisso ai campi del PD; ad esempio `current->pid` restituisce il PID del processo corrente.

Un altro vantaggio di memorizzare il PD con lo stack si ha nei sistemi multiprocessore: il processo corrente per ogni CPU può essere derivato dallo stack. Vecchie versioni di Linux non associavano stack del kernel e PD. Erano perciò costrette a introdurre una variabile statica globale chiamata `current` per identificare il PD. Nei sistemi multiprocessore era necessario definire `current` come array, con un elemento per CPU.

Doubly linked lists

E' necessario ora introdurre una struttura dati speciale che implementa le liste a collegamento doppio (doubly linked list – DLL). Per gestire le strutture lista occorre un insieme di operazioni fondamentali; inizializzazione, inserimento ed eliminazione di elementi, ricerca entro la lista e così via. Sarebbe un inutile dispendio di energie dover ripetere le funzioni per ogni lista.

Perciò il kernel definisce la struttura dati `list_head`, i cui unici campi `next` e `prev` rappresentano i puntatori verso e da un elemento generico di una DLL. E' da notare che i puntatori della `list_head` contengono gli indirizzi dei campi `list_head` e non dell'intera struttura dati che li contiene.

Una nuova lista viene creata dalla macro `LIST_HEAD(list_name)`. Essa dichiara una nuova variabile chiamata `list_name` di tipo `list_head`, che rappresenta un segnaposto per l'inizio di una nuova lista, e inizializza i campi `next` e `prev` in modo che puntino alla variabile `list_name` stessa.

Diverse macro e funzioni implementano le operazioni fondamentali:

`list_add(n,p)`: inserisce un elemento puntato da `n` subito dopo l'elemento puntato da `p`; per inserirlo all'inizio della lista, si attribuisce a `p` l'indirizzo del primo elemento della lista;

`list_add_tail(n,p)`: inserisce un elemento dopo `p`;

`list_del(p)`: elimina l'elemento puntato da `p`;

`list_empty(p)`: controlla se la lista di cui `p` contiene l'indirizzo del primo elemento è vuota;

`list_entry(p,t,m)`: restituisce l'indirizzo della struttura di tipo `t` nella quale il campo `list_head` ha il nome `m` e l'indirizzo `p`;

`list_for_each(p,h)`: scandisce gli elementi della lista di cui `h` è l'indirizzo del primo elemento. In ogni iterazione viene restituito in `p` un puntatore alla struttura `list_head` dell'elemento della lista;

`list_for_each_entry(p,h,m)`: simile alla precedente, ma restituisce un puntatore alla struttura dati che comprende il campo `list_head`.

Il kernel 2.6 supporta un altro tipo di DLL che differisce dalla precedente perché non è circolare; viene usata soprattutto nelle tabelle di hash, dove lo spazio è importante, e dove non lo è invece trovare l'elemento finale in un tempo costante. L'inizio della lista è contenuto in una struttura di tipo `hlist_head`, che è semplicemente un puntatore al primo elemento della lista (NULL se la lista è vuota). Ogni elemento è rappresentato da una struttura `hlist_node`, che comprende un puntatore `next` all'elemento successivo e un puntatore `pprev` al campo `next` dell'elemento precedente. Poiché la lista non è circolare, il campo `pprev` del primo elemento e il campo `next` dell'ultimo hanno valore NULL. La lista può essere gestita con funzioni simili a quelle elencate in precedenza: `hlist_add_head()`, `hlist_del()`, `hlist_empty()`, `hlist_entry_list()`, `hlist_foreach_entry()` ecc.

La lista dei processi

Il primo esempio di una DLL è la lista dei processi, che lega fra loro tutti i PD. Ogni struttura `task_struct` contiene un campo `tasks` di tipo `list_head`, i cui campi `prev` e `next` puntano alle `task_struct` precedente e successiva.

Il primo elemento della lista è il descrittore `init_task task_struct`: è il PD del cosiddetto processo 0 o swapper. Il campo `tasks->prev` di `init_task` punta al campo `tasks` dell'ultimo PD inserito in lista.

Le macro `SET_LINKS` e `REMOVE_LINKS` sono usate per inserire o rimuovere un PD della lista; queste macro si occupano anche delle relazioni genitore-figlio tra processi.

Un'altra macro utile, `for_each_process`, esamina l'intera lista. E' definita come:

```
#define for_each_process(p) \
    for(p=&init_task; (p=list_entry((p)->tasks.next, \
                                   struct task_struct, tasks) \
                                   ) != &init_task;)
```

La macro è la dichiarazione di controllo del ciclo, dopo la quale il programmatore del kernel scrive il ciclo stesso. Da notare che il PD `init_task` gioca solo il ruolo di primo elemento. La macro inizia dall'elemento successivo ad `init_task` e continua fino a raggiungere di nuovo `init_task` (la lista è circolare). Ad ogni iterazione, la variabile passata come argomento della macro contiene l'indirizzo del PD attualmente scandito, quello restituito dalla macro `list_entry`.

Lista dei processi TASK_RUNNING

Quando cerca un processo per l'esecuzione, la CPU deve considerare solo quelli eseguibili, cioè nello stato `TASK_RUNNING`. Le prime versioni di Linux mettevano tutti i processi eseguibili nella stessa coda detta *runqueue*. Poiché era troppo dispendioso mantenere ordinata la lista in base alle priorità dei processi, i primi scheduler dovevano scandire l'intera coda per selezionare il processo "migliore" per l'esecuzione.

Linux 2.6 implementa diversamente la runqueue. L'obiettivo è di consentire allo scheduler la scelta in tempo costante, indipendentemente dal numero di processi eseguibili.

Il trucco consiste nel dividere la runqueue in varie liste, una per ogni priorità. Ogni `task_struct` possiede un campo `run_list` di tipo `list_head`. Se la priorità del processo è uguale a `k` (con valori da 0 a 139), il campo `run_list` collega i descrittori dei processi entro la lista di quelli eseguibili che hanno priorità `k`. Nei sistemi multiprocessore ogni CPU ha la sua runqueue. Questo è il classico esempio di come si rendono le strutture più complicate per migliorare le prestazioni: per rendere più efficiente lo scheduler, la runqueue è stata

divisa in 140 liste diverse.

Per mantenere ogni runqueue occorre memorizzare molti dati; le strutture fondamentali rimangono però le liste dei descrittori che appartengono alla runqueue; esse sono implementate da una singola struttura `prio_array_t`, i cui campi sono i seguenti:

tipo	campo	descrizione
int	<code>nr_active</code>	numero dei PD collegati nella lista

tipo	campo	descrizione
unsigned long	bitmap	bitmap delle priorità; ogni bit è attivato se la lista della priorità corrispondente non è vuota
struct list_head[140]	queue	I primi elementi delle 140 liste

La funzione `enqueue_task(p, array)` inserisce un PD in una delle liste; il codice è simile a questo:

```
list_add_tail(&p->run_list, &array->queue[p->prio]);
__set_bit(p->prio, array->bitmap);
array->nr_active++;
p->array = array;
```

Il campo `prio` del PD contiene la priorità dinamica del processo, mentre il campo `array` è un puntatore alla struttura `prio_array_t` della propria runqueue corrente.

La funzione `dequeue_task(p, array)` rimuove un PD dalla lista.

Relazioni tra processi

I processi hanno relazioni genitore-figlio; quando un processo crea diversi figli, essi sono fratelli. Vari campi nel descrittore rappresentano queste relazioni, e sono elencati di seguito, riferendoli a un processo P. I processi 0 e 1 sono creati dal kernel; il processo 1 (`init`) è il padre di tutti i processi.

- `real_parent`: punta al PD del processo che ha creato P oppure al PD di `init` se il genitore non esiste più (è il caso di un utente che avvia un processo in background e poi chiude la shell);
- `parent`: punta al genitore di P, cioè al processo che riceve un segnale quando il figlio termina. In generale coincide con `real_parent`; può essere diverso nel caso in cui un processo chiama la `ptrace()` per monitorare P.
- `children`: il primo elemento della lista che contiene i figli di P
- `sibling`: i puntatori al successivo e precedente elemento della lista dei processi fratelli.

Un processo può essere anche il leader di un gruppo di processi, di una sessione di login, di un gruppo di thread oppure può eseguire il debugging su un altro. I campi che stabiliscono queste informazioni sono i seguenti:

- `group_leader`: puntatore al PD del leader del gruppo a cui appartiene P
- `signal->pgrp`: PID del leader del gruppo
- `tgid`: leader del thread group
- `signal->session`: PID del leader della sessione di login di P
- `ptrace_children`: il primo elemento di una lista che contiene tutti i figli di P monitorati da un debugger
- `ptrace_list`: puntatori agli elementi successivo e precedente della lista dei `real_parents` dei processi

monitorati (usato quando P è oggetto di una ptrace()).

Tabella pidhash e liste concatenate

Il kernel deve poter ricavare il puntatore al PD che corrisponde ad un dato PID, ad esempio quando deve elaborare una chiamata di sistema kill(). Quando un processo P1 vuole inviare un segnale a P2, chiama la kill() specificando il PID di P2 come parametro. Il kernel ricava il puntatore al PD in base al PID fornito, poi estrae il puntatore alla struttura che contiene i segnali sospesi dal PD di P2.

Esaminare sequenzialmente tutta la lista dei processi e controllare i campi pid è possibile ma poco efficiente. Per sveltire la ricerca, sono state introdotte 4 tabelle di hash; sono 4 perché il PD ha diversi campi con diversi tipi di PID che richiedono ognuno una tabella di hash.

tipo di tabella hash	nome campo	descrizione
PIDTYPE_PID	pid	PID del processo
PIDTYPE_TGID	tgid	PID del leader del gruppo di thread
PIDTYPE_PGID	pgrp	PID del leader del gruppo
PIDTYPE_SID	session	PID del leader di sessione

Le 4 tabelle di hash sono allocate dinamicamente dal kernel durante la fase di inizializzazione, e il loro indirizzo è memorizzato nell'array pid_hash. La dimensione di ogni tabella dipende dalla quantità di RAM; in sistemi con 512 MB, ogni tabella è contenuta in 4 frame di pagina e comprende 2048 entry.

Il PID è trasformato in un indice della tabella usando la macro pid_hashfn così definita:

```
#define pid_hashfn(x) hash_long((unsigned long) x, pidash_shift)
```

La variabile pidhash_shift contiene la lunghezza in bit di un indice di tabella (11 nell'esempio). La funzione hash_long() è usata in molte funzioni di hash ed equivale a:

```
unsigned long hash_long(unsigned long val, unsigned int bits) {  
    unsigned long hash = val * 0x9e370001UL;  
    return hash >> (32 - bits);  
}
```

Poiché il valore di pidhash_shift è 11, i valori di pid_hashfn sono compresi tra 0 e $2^{11} - 1 = 2047$.

Ci si può chiedere da dove provenga il valore 0x9e370001 (2.654.404.609 decimale). La funzione di hash si basa sul prodotto dell'indice con un numero tanto grande da provocare un overflow; il valore che rimane nella variabile a 32 bit può essere considerato risultato di una operazione modulo. Knuth ha suggerito che si ottengono buoni risultati con un valore primo in rapporto aureo con 2^{32} . Ora, 2.654.404.609 è primo rispetto a $2^{32} \times (\sqrt{5} - 1)/2$, e può essere moltiplicato facilmente con addizioni e shift di bit essendo uguale a

$$2^{31} + 2^{29} - 2^{25} + 2^{22} - 2^{19} - 2^{16} + 1$$

Una funzione di hash non assicura una corrispondenza uno a uno tra PID e indice di tabella. Due PID che

generano un hash nello stesso indice collidono. Per gestire le collisioni, Linux usa il concatenamento; ogni entry di tabella è il primo elemento di una lista a doppio collegamento di PD che collidono. Ad esempio, i due processi con PID 2890 e 29384 si trovano collegati nella tabella al 200° elemento, mentre il processo con PID 29385 si trova collegato al 1466° elemento.

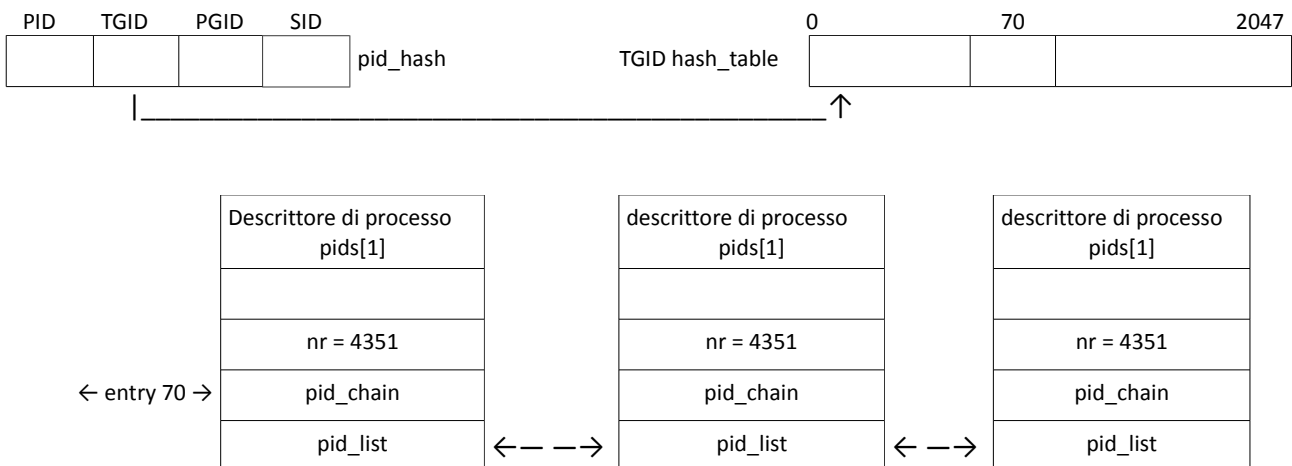
Questo sistema è preferibile alla trasformazione lineare da PID ad indici di tabella perché, in un dato istante, il numero di processi è solitamente molto inferiore a 32.768 (il numero massimo di PID ammessi dal sistema) e sarebbe uno spreco definire una tabella con 32.768 entry, la maggior parte delle quali vuote.

Le strutture usate nelle tabelle di hash dei PID devono tenere traccia delle relazioni di parentela tra processi. Se, ad esempio, il kernel deve trovare tutti i processi che appartengono ad un dato gruppo di thread, cioè quelli che hanno lo stesso valore di tgid, consultando la tabella di hash ricava il descrittore del leader del gruppo. Per trovare velocemente gli altri componenti, deve conservare una lista dei processi per ogni gruppo.

Le strutture dati delle tabelle di hash risolvono questo problema, perché consentono di definire una lista di processi per ogni PID incluso in tabella. La struttura fondamentale è un array di 4 strutture pid incorporate nel campo pids del descrittore di processo. I campi della struttura pid sono i seguenti:

tipo	nome	descrizione
int	nr	PID
struct hlist_node	pid_chain	collegamenti agli elementi precedente e successivo della catena degli hash
struct list_head	pid_list	primo elemento della lista per PID

Tabelle hash



Un esempio basato sulla tabella hash `PIDTYPE_TGID`: la seconda entry dell'array `pid_hash` contiene l'indirizzo della tabella hash `TGID`. Nella lista che si origina dalla 71° entry sono collegati due PD con i valori di PID 246 (non raffigurato) e 4351. I valori di PID sono memorizzati nel campo `nr` della struttura `pid` contenuta nel PD (per inciso, dato che il numero di thread group coincide con il PID del leader, questo numero è contenuto anche nel campo `pid` del PD). Si consideri la lista per-PID del gruppo di thread 4351: l'inizio della lista è contenuto nel campo `pid_list` del PD, mentre i collegamenti agli elementi precedente e successivo

sono contenuti nei campi `pid_list` di ogni elemento della lista.

Le macro e funzioni seguenti sono usate per gestire le tabelle hash dei PID:

- `do_each_task_pid(nr, type, task)` e `while_each_task_pid(nr, type, task)`: segnano inizio e fine di un ciclo do-while che compie una iterazione sulla lista per-PID associata al PID `nr` di tipo `type`; `task` punta al PD dell'elemento attualmente scandito.
- `find_task_by_pid_type(type, nr)`: cerca il processo che ha PID `nr` nella tabella hash di tipo `type`; restituisce un puntatore a PD oppure NULL;
- `find_task_by_pid(nr)`: uguale a `find_task_by_pid_type(PIDTYPE_PID, nr)`;
- `attach_pid(task, type, nr)`: inserisce il PD puntato da `task` nella tabella hash di tipo `type` in accordo col PID numero `nr`; se un PD con PID `nr` è già nella tabella hash, la funzione inserisce semplicemente `task` nella lista per PID del processo già presente;
- `detach_pid(task, type)`: rimuove il PD puntato da `task` dalla lista per-PID di tipo `type` a cui il PD appartiene. Se la lista non rimane vuota, la funzione termina. Altrimenti la funzione rimuove il PD dalla tabella hash di tipo `type`; infine, se il PID non compare in nessun'altra tabella hash, la funzione azzerava il bit corrispondente nella bitmap dei PID, in modo che il numero possa essere riciclato;
- `next_thread(task)`: restituisce l'indirizzo del PD del processo "leggero" che segue `task` nella tabella hash di tipo `PIDTYPE_TGID`. Poiché la lista è circolare, quando viene applicata ad un processo regolare, restituisce l'indirizzo del processo stesso.

Organizzazione dei processi

Le liste della runqueue raggruppano tutti i processi nello stato `TASK_RUNNING`. I processi in altri stati richiedono diverso trattamento, con Linux che si trova a scegliere tra due opzioni:

- i processi nello stato `TASK_STOPPED`, `EXIT_ZOMBIE` e `EXIT_DEAD` non sono collegati in liste specifiche. Non c'è necessità di raggrupparli perché si accede a loro solo attraverso il PID o per mezzo delle liste dei processi figli di un particolare genitore.
- I processi nello stato `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE` sono suddivisi in varie classi, in base ad eventi specifici. In questo caso lo stato del processo non fornisce sufficienti indicazioni per accedere ad esso in breve tempo, per cui è necessario aggiungere altre liste: le code di attesa (wait queues - WQ).

Wait queues

Le WQ hanno diversi scopi nel kernel, in particolare nella gestione degli interrupt, nella sincronizzazione dei processi, nella gestione del tempo. Spesso un processo deve attendere il verificarsi di certi eventi, come il completamento di una operazione su disco, il rilascio di una risorsa o lo scadere di un intervallo di tempo. Le WQ implementano l'attesa condizionata agli eventi: un processo in attesa viene inserito nella lista adatta e cede il controllo. Quindi una WQ rappresenta un insieme di processi quiescenti, che vengono risvegliati dal kernel quando una certa condizione si avvera.

Le WQ sono implementate con liste a doppio collegamento, i cui elementi includono puntatori a PD. Ogni lista è identificata da un elemento iniziale, una struttura di tipo `wait_queue_head_t`:

```

struct wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};

```

Poi ché le WQ sono modificate dai gestori degli interrupt e da funzioni del kernel, devono essere protette da accessi concorrenti che possono portare a risultati imprevedibili. La sincronizzazione è assicurata dallo spinlock lock dell'elemento iniziale. Il campo task_list è l'inizio della lista dei processi in attesa.

Gli elementi di una WQ sono di tipo wait_queue_t:

```

struct __wait_queue {
    unsigned int flag;
    struct task_struct *task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;

```

Ogni elemento rappresenta un processo quiescente; il suo PD è memorizzato nel campo task. task_list contiene i puntatori che collegano l'elemento agli altri processi in attesa dello stesso evento.

Non sempre è conveniente risvegliare tutti i processi di una WQ. Se due o più processi sono in attesa per un accesso esclusivo ad una risorsa, ha più senso risvegliarne solo uno. Esso acquisisce la risorsa mentre gli altri restano inattivi. Questo previene il problema detto "thundering herd", per cui vari processi sono risvegliati per competere per una risorsa che solo uno può acquisire, col risultato che gli altri devono ritornare inattivi.

Ci sono due tipi di processi quiescenti: i *processi esclusivi* (caratterizzati dal valore 1 nel campo flag del corrispondente elemento della coda) sono selettivamente risvegliati dal kernel, mentre i *non esclusivi* (valore 0 del campo flag) sono sempre risvegliati dal kernel quando avviene l'evento. Un processo in attesa di una risorsa che può essere acquisita da un solo utente alla volta è un tipico esempio di processo esclusivo. I processi in attesa di un evento che può interessarli tutti sono non esclusivi. Si consideri un gruppo di processi che sta aspettando il trasferimento di alcuni blocchi dal disco; appena viene completata l'operazione, tutti i processi possono essere risvegliati. Il campo func di un elemento di WQ viene usato per specificare come un processo deve essere risvegliato.

Gestione delle wait queues

La macro DECLARE_WAIT_QUEUE_HEAD(name) dichiara staticamente una wait queue head con nome name, e inizializza i valori di lock e task_list_field. La funzione init_waitqueue_head() viene usata per inizializzare una variabile dello stesso tipo allocata dinamicamente.

La funzione init_waitqueue_entry(q, p) inizializza una struttura q di tipo wait_queue_t in questo modo:

```

q->flags = 0;
q->task = p;
q->func = default_wake_function;

```

Il processo non esclusivo p è risvegliato da default_wake_function(), un wrapper per try_to_wake_up(). In alternativa la macro DEFINE_WAIT dichiara una nuova variabile wait_queue_t e la inizializza col PD del processo al momento in esecuzione e l'indirizzo della funzione autoremove_wake_function(). Essa chiama default_wake_function() per risvegliare il processo, poi rimuove l'elemento dalla lista di WQ. Uno

sviluppatore del kernel può anche definire una funzione personalizzata per risvegliare un processo, inizializzando l'elemento della WQ con `init_waitqueue_func_entry()`.

Quando un elemento viene creato, deve essere inserito in una WQ. La funzione `add_wait_queue()` inserisce un processo non esclusivo nella prima posizione di una lista di WQ. La funzione `add_wait_queue_exclusive()` inserisce un processo esclusivo all'ultimo posto della lista. La funzione `remove_wait_queue()` rimuove un processo dalla lista; la `waitqueue_active()` controlla se una data lista è vuota.

Un processo che vuole attendere una specifica condizione può chiamare una delle seguenti funzioni:

`sleep_on()` opera sul processo corrente:

```
void sleep_on(wait_queue head_t *wq) {
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq, &wait); /* wait points to wait queue head */
    schedule();
    remove_wait_queue(wq, &wait);
}
```

La funzione imposta il processo a `TASK_UNINTERRUPTIBLE` e lo inserisce nella WQ specificata. Poi chiama lo scheduler; quando il processo viene risvegliato, la funzione lo toglie dalla WQ

`interruptible_sleep_on()` è identica alla precedente, ma imposta il processo a `TASK_INTERRUPTIBLE`; in questo modo può essere risvegliato anche da un segnale.

`sleep_on_timeout()` e `interruptible_sleep_on_timeout()` sono simili alle precedenti, salvo permettere al chiamante di definire un intervallo di tempo dopo il quale il processo verrà risvegliato. Per questo motivo chiamano `schedule_timeout()` invece di `schedule()`.

`prepare_to_wait()`, `prepare_to_wait_exclusive()` e `finish_wait()` introdotte in Linux 2.6, offrono un altro modo di inserire il processo corrente in una WQ. Vengono usate come segue:

```
DEFINE_WAIT(wait);
prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
...
if(! condition)
    schedule();
finish_wait(&wq, &wait);
```

Le funzioni pongono il processo nello stato specificato dal terzo parametro, impostano di conseguenza il flag esclusivo e infine inseriscono l'elemento `wait` nella lista che ha come inizio `wq`.

Appena il processo viene risvegliato, esegue la `finish_wait()` che pone il processo nello stato `TASK_RUNNING` (nel caso in cui la condizione per il risveglio si verifichi prima della chiamata a `schedule()`) e poi toglie l'elemento dalla lista di WQ.

Le macro `wait_event()` e `wait_event_interruptible()` inseriscono il processo corrente in una WQ fino a quando si verifica una certa condizione. Il codice è essenzialmente questo:

```
DEFINE_WAIT(__wait);
for(;;) {
```

```

        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
        if(condition)
            break;
        schedule();
    }
    finish_wait(&wq, &__wait);

```

Qualche considerazione sulle funzioni descritte: le funzioni tipo `sleep_on()` non possono essere impiegate nelle situazioni comuni in cui bisogna verificare una condizione e mettere a riposo atomicamente un processo; poiché sono fonte di *race conditions*, il loro uso è sconsigliato.

Per inserire un processo esclusivo in una WQ, il kernel deve usare la `prepare_to_wait_exclusive()` (oppure chiamare `add_wait_queue_exclusive()` direttamente); ogni altra funzione inserisce il processo come non esclusivo. Inoltre finché non vengono usate `DEFINE_WAIT` o `finish_wait()`, il kernel deve rimuovere l'elemento della WQ dalla lista dopo che il processo è stato risvegliato.

Il kernel risveglia i processi nelle WQ ponendoli nello stato di `TASK_RUNNING` per mezzo di una delle macro seguenti: `wake_up`, `wake_up_nr`, `wake_up_all`, `wake_up_interruptible`, `wake_up_interruptible_nr`, `wake_up_interruptible_all`, `wake_up_interruptible_sync` e `wake_up_locked`. Si può comprenderne lo scopo dal nome:

- tutte le macro considerano i processi nello stato `TASK_INTERRUPTIBLE`; se non hanno il termine “interruptible” nel nome, considerano anche lo stato `TASK_UNINTERRUPTIBLE`;
- tutte risvegliano tutti i processi non esclusivi;
- le macro il cui nome include “nr” risvegliano nr processi esclusivi; nr è fornito come parametro, quelle il cui nome comprende “all” risvegliano tutti i processi esclusivi, infine quelle il cui nome non include né “nr” né “all” risvegliano un solo processo esclusivo;
- le macro il cui nome non include “sync” controllano che la priorità del processo risvegliato sia maggiore di quella del processo corrente e chiamano `schedule()` se necessario; questi controlli non sono fatti dalle macro che non hanno “sync” nel nome; di conseguenza l'esecuzione di un processo con priorità alta può essere leggermente ritardata;
- `wake_up_locked` è simile a `wake_up`, tranne che viene chiamata quando lo spinlock in `wait_queue_head_t` è già stato acquisito.

La macro `wake_up` è sostanzialmente equivalente a questo codice:

```

void wake_up(wait_queuehead_t *q) {
    struct list_head *tmp;
    wait_queue_t *curr;
    list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if(curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE, 0, NULL) && curr->flags)
            break;
    }
}

```

La macro `list_for_each` scandisce ogni elemento della lista `q->task_list`, cioè ogni processo nella WQ. Per ogni elemento la macro `list_entry` calcola l'indirizzo della corrispondente variabile `wait_queue_t`. Il campo `func` di questa variabile contiene l'indirizzo della funzione di `wake_up`, che tenta di risvegliare il processo identificato dal campo `task`. Se un processo è stato effettivamente risvegliato, la funzione restituisce 1 e se il

processo è esclusivo (curr->flags uguale a 1) il ciclo termina. Poiché tutti i processi non esclusivi sono sempre all'inizio della lista a doppio collegamento e tutti quelli esclusivi sono alla fine, la funzione risveglia tutti i processi non esclusivi e un esclusivo se è presente¹.

Limiti di risorse dei processi

Ogni processo ha un insieme di *limiti di risorse* che specificano l'ammontare delle risorse di sistema che può impiegare. Essi impediscono ad un utente di esaurire le risorse del sistema

I limiti del processo corrente sono memorizzati nel campo `current->signal->rlim`, cioè in un campo del descrittore dei segnali del processo. Il campo è un array di strutture di tipo `rlimit`, una per ogni tipo di limite:

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

- `RLIMIT_AS`: dimensione massima dello spazio di indirizzamento in byte. Il kernel controlla il limite quando un processo chiama `malloc()` o un'altra funzione per ampliare lo spazio.
- `RLIMIT_CORE`: dimensione massima del file core in byte. Il kernel controlla il limite quando un processo abortisce prima di creare un file core nella directory corrente. Se il limite è 0 il kernel non crea file core.
- `RLIMIT_CPU`: massimo tempo di CPU in secondi. Se il processo lo supera, il kernel manda un segnale `SIGXCPU` e, se il processo non termina, un segnale `SIGKILL`.
- `RLIMIT_DATA`: massima dimensione di heap in byte. Il kernel la controlla prima di espandere l'heap.
- `RLIMIT_FSIZE`: massima dimensione di un file in byte. Se un processo tenta di aumentare la dimensione di un file oltre il limite, il kernel manda un segnale `SIGXFSZ`.
- `RLIMIT_LOCKS`: numero massimo di lock di file (non applicato al momento).
- `RLIMIT_MEMLOCK`: dimensione massima della memoria non soggetta a swap, in byte. Il kernel controlla questo valore quando un processo tenta di creare un lock su un frame di pagina usando le chiamate di sistema `mlock()` o `mlockall()`.
- `RLIMIT_MSGQUEUE`: numero massimo di byte nelle code dei messaggi POSIX.
- `RLIMIT_NOFILE`: massimo numero di descrittori di file aperti. Il kernel lo controlla quando viene aperte un nuovo file o duplicato un descrittore.
- `RLIMIT_NPROC`: numero massimo di processi di cui un utente può essere proprietario.
- `RLIMIT_RSS`: numero massimo di frame di pagina posseduti dal processo (non applicato al momento).
- `RLIMIT_SIGPENDING`: massimo numero di segnali sospesi per processo.
- `RLIMIT_STACK`: dimensione massima dello stack in byte. Il kernel controlla il limite prima di aumentare lo stack in modalità utente.

¹ In realtà è molto raro che la `wq` comprenda sia processi esclusivi che non esclusivi.

Il campo `rlim_cur` è il limite attuale della risorsa; ad esempio `current->signal->rlim[RLIMIT_CPU].rlim_cur` rappresenta il valore attuale del limite di tempo della CPU.

Il campo `rlim_max` è il valore massimo ammesso per il limite della risorsa. Usando le chiamate di sistema `getrlimit()` e `setrlimit()` un utente può aumentare `rlim_cur` fino a `rlim_max`. Solo `root` può aumentare `rlim_max` o impostare `rlim_cur` ad un valore maggiore del massimo.

Molti limiti contengono `RLIM_INFINITY (0xffffffff)`, che significa nessun limite imposto, a parte quelli fisici legati alla macchina (RAM, spazio su disco ecc.). Tuttavia `root` può imporre limiti inferiori. Quando un utente esegue il login nel sistema, il kernel crea un processo di cui è proprietario il `superuser`, che può chiamare `setrlimit()` per diminuire `rlim_max` e `rlim_cur`. Lo stesso processo poi esegue una shell di login e diviene di proprietà dell'utente. Ogni nuovo processo creato dall'utente eredita dal genitore l'array `rlim`.

Commutazione di processo

Il kernel deve poter sospendere l'esecuzione del processo corrente della CPU e riprendere l'esecuzione di altri processi precedentemente sospesi. Questa operazione va sotto i nomi di commutazione di processo, commutazione di task, commutazione di contesto.

Contesto hardware

Ogni processo ha il suo spazio di indirizzamento ma deve condividere i registri della CPU. Così prima di riprendere l'esecuzione di un processo, il kernel deve assicurarsi che ogni registro contenga il valore che aveva prima che il processo fosse sospeso.

L'insieme dei valori da caricare nei registri prima che il processo riprenda l'esecuzione è chiamato contesto hardware. Il contesto hardware è un sottoinsieme del contesto di esecuzione del processo, che include tutte le informazioni necessarie per l'esecuzione. In Linux una parte del contesto hardware è contenuto nel PD, mentre il resto è salvato nello stack in modalità del kernel.

Nella descrizione che segue, la variabile locale `prev` si riferisce al PD del processo interrotto e `next` si riferisce al PD di quello che deve prenderne il posto. Commutazione di processo è l'operazione di salvataggio del contesto hardware di `prev` e della sua sostituzione con quello di `next`. Poiché le commutazioni di contesto sono frequenti, è importante ridurre al minimo il tempo impiegato in questa operazione.

Le vecchie versioni di Linux traevano vantaggio dal supporto hardware offerto dall'architettura 80x86 ed eseguivano una commutazione di contesto attraverso una istruzione di salto (far `jmp`²) al selettore del Task State Segment Descriptor del nuovo processo. Quando esegue questa istruzione, la CPU esegue una commutazione di contesto hardware salvando automaticamente il vecchio e sostituendolo col nuovo. Linux 2.6 usa il software per il cambio di contesto per varie ragioni:

- la commutazione passo a passo attraverso istruzioni `mov` consente un miglior controllo sulla validità dei dati. In particolare è possibile controllare il valore dei registri di segmento `es` e `ds` che potrebbero essere stati alterati in modo doloso. Questo controllo non è possibile usando una singola istruzione `far jmp`.
- L'impegno di tempo per i due metodi è lo stesso; non è possibile ottimizzare una commutazione hardware, mentre c'è sempre spazio per migliorare il codice.

² `far jmp` modifica i registri `cs` e `eip`, mentre `jmp` modifica solo `eip`

La commutazione di processo avviene solo in modalità del kernel. I registri usati in modalità utente sono già stati salvati sullo stack in modalità del kernel prima che avvenga la commutazione, compresi anche `ss` e `esp` che specificano l'indirizzo del puntatore allo stack in modalità utente.

Task State Segment

L'architettura 80x86 include un tipo speciale di segmento, chiamato Task State Segment (TSS), per memorizzare il contesto hardware. Anche se Linux non sfrutta la commutazione hardware, deve inizializzare un TSS per ogni CPU per due motivi principali:

- quando una CPU passa da modalità utente a kernel, estrae l'indirizzo dello stack in modalità del kernel dal TSS
- quando un processo in modalità utente tenta di accedere ad una porta di I/O per mezzo di una istruzione `in` oppure `out`, la CPU può avere la necessità di consultare la bitmap dei permessi di I/O conservata nel TSS per verificare che il processo sia abilitato a indirizzare la porta.

Più precisamente quando un processo esegue una istruzione `in` oppure `out`, l'unità di controllo esegue le seguenti operazioni:

- controlla il campo di 2 bit `IOPL` nel registro `eflag`. Se è impostato a 3, l'unità di controllo esegue le istruzioni di I/O, altrimenti esegue i controlli successivi;
- accede al registro `tr` per determinare il TSS corrente e quindi la bitmap dei permessi di I/O;
- controlla il bit della Bitmap dei Permessi di I/O corrispondente alla porta di I/O specificato nell'istruzione di I/O. Se è azzerato, l'istruzione viene eseguita; altrimenti l'unità di controllo genera un'eccezione di "General protection".

```
struct tss_struct {
    unsigned short back_link, __blh;
    unsigned long esp0;
    unsigned short ss0, __ss0h;
    unsigned long esp1;
    unsigned short ss1, __ss1h; /* ss1 is used to cache MSR_IA32_SYSENTER_CS */
    unsigned long esp2;
    unsigned short ss2, __ss2h;
    unsigned long __cr3;
    unsigned long eip;
    unsigned long eflags;
    unsigned long eax, ecx, edx, ebx;
    unsigned long esp;
    unsigned long ebp;
    unsigned long esi;
    unsigned long edi;
    unsigned short es, __esh;
    unsigned short cs, __csh;
    unsigned short ss, __ssh;
    unsigned short ds, __dsh;
    unsigned short fs, __fsh;
    unsigned short gs, __gsh;
    unsigned short ldt, __ldth;
```

```

unsigned short trace, io_bitmap_base;
/*
 * The extra 1 is there because the CPU will access an
 * additional byte beyond the end of the IO permission
 * bitmap. The extra byte must be all 1 bits, and must
 * be within the limit.
 */
unsigned long io_bitmap[IO_BITMAP_LONGS + 1];
/*
 * Cache the current maximum and the last task that used the bitmap:
 */
unsigned long io_bitmap_max;
struct thread_struct *io_bitmap_owner;
/*
 * pads the TSS to be cacheline-aligned (size is 0x100)
 */
unsigned long __cacheline_filler[35];
/*
 * .. and then another 0x100 bytes for emergency kernel stack
 */
unsigned long stack[64];
} __attribute__((packed));

```

La struttura `tss_struct` descrive il formato del TSS. L'array `init_tss` contiene un TSS per ogni CPU del sistema. Ad ogni commutazione di processo il kernel aggiorna alcuni campi del TSS in modo che l'unità di controllo della CPU corrispondente possa recuperare le informazioni di cui ha bisogno. Perciò il TSS riflette il privilegio del processo corrente nella CPU, ma non c'è bisogno di mantenere TSS per processi non in esecuzione.

Ogni TSS ha il proprio Descrittore di TSS di 8 byte. Questo descrittore include il campo Base di 32 bit che punta all'indirizzo iniziale di TSS e un campo Limit di 20 bit. Il flag S del descrittore di TSS è azzerato per indicare che si tratta di un Segmento di Sistema.

Il campo Type è impostato a 9 o a 11 per indicare che il segmento è un TSS. Nel progetto originale Intel ogni processo dovrebbe far riferimento al proprio TSS; il secondo bit meno significativo del campo Type è chiamato Busy bit; è impostato a 1 se il processo sta per essere eseguito dalla CPU, 0 altrimenti. Nel progetto Linux, c'è solo un TSS per CPU e il Busy bit è sempre a 1.

I descrittori di TSS creati da Linux sono contenuti nella GDT. Il registro `tr` contiene il Selettore del Descrittore del TSS. Il registro include anche due campi nascosti non programmabili: Base e Limit del Descrittore di TSS. In questo modo il processore può indirizzare il TSS direttamente senza dover consultare la GDT.

Il campo thread

Ad ogni commutazione di processo, il contesto hardware del processo sostituito deve essere salvato in qualche posto. Non può essere salvato nel TSS, come nel progetto originale Intel, perché Linux usa un solo TSS per CPU, invece di uno per processo.

Perciò ogni PD include un campo chiamato `thread` di tipo `thread_struct` nel quale il kernel salva il contesto hardware. Questa struttura include campi per la maggior parte dei registri, eccetto quelli generali come `eax`, `ebx`, ecc. che vengono salvati nello stack in modalità del kernel.

Esecuzione della commutazione di processo

La commutazione di processo può avvenire solo in un momento preciso; alla chiamata della funzione `schedule()`. Consiste di due fasi principali:

- cambio della GDT per aprire un nuovo spazio di indirizzamento;
- commutazione dello stack in modalità del kernel e del contesto hardware, che fornisce tutte le informazioni necessarie al kernel per eseguire il nuovo processo, inclusi i registri della CPU.

Come in precedenza, `prev` contiene l'indirizzo del PD del processo da sostituire e `next` quello del PD del processo da eseguire; `prev` e `next` sono variabili locali della funzione `schedule()`.

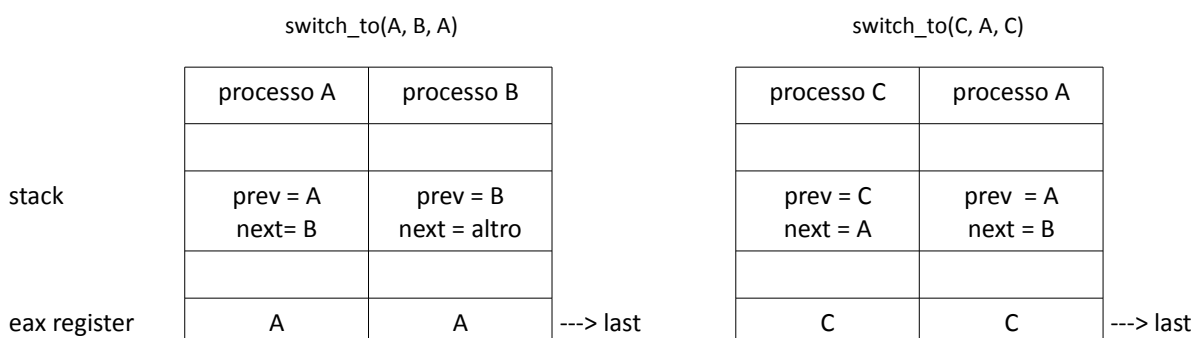
La macro `switch_to`

La seconda fase della commutazione di processo è attuata dalla macro `switch_to`. Prima di tutto, la macro ha tre parametri chiamati `prev`, `next` e `last`. I primi due sono solo segnaposti per le variabili locali `prev` e `next`, che hanno il significato esposto sopra. In ogni cambio di contesto sono coinvolti 3 processi. Se deve avvenire il cambio tra A e B, `prev` punta al descrittore di A, `next` a quello di B. Appena `switch_to` interrompe A, il suo flusso di esecuzione viene bloccato.

Più tardi, quando il kernel vuole riavviare A, deve sostituire un altro processo C, in generale diverso da B, eseguendo `switch_to` con `prev` che punta a C e `next` che punta ad A. Quando A riprende il suo flusso di esecuzione, trova il suo vecchio stack in modalità del kernel, in modo che la variabile locale `prev` punta al descrittore di A e `next` al descrittore di B. Lo scheduler, che ora viene eseguito per conto di A, ha perso ogni riferimento a C. Questo riferimento però torna utile per completare il cambio di contesto.

Il terzo parametro di `switch_to`, `last`, è un parametro di output che specifica la locazione in cui la macro scrive l'indirizzo del PD di C (questo viene fatto dopo che A ha ripreso l'esecuzione). Prima della commutazione, la macro salva nel registro `eax` il contenuto della variabile identificata dal primo parametro di input `prev` – vale a dire la variabile locale `prev` collocata nello stack in modalità del kernel di A. Dopo la commutazione, quando A riprende l'esecuzione, la macro scrive il contenuto del registro `eax` nella locazione di memoria di A identificata dal terzo parametro `last`. Poiché il registro non cambia attraverso la commutazione, questa locazione riceve l'indirizzo del descrittore di C. Nella versione attuale di `schedule()`, `last` identifica la variabile locale `prev` di A, in modo che `prev` viene sovrascritto con l'indirizzo di C.

Il contenuto degli stack in modalità del kernel dei processi A, B e C sono mostrati in figura, insieme al valore di `eax`; viene mostrato il valore della variabile `prev` *prima* che il valore sia sovrascritto dal contenuto di `eax`.



La macro `switch_to` è codificata in linguaggio assembly inline esteso che rende piuttosto complessa la lettura: infatti si riferisce ai registri con una notazione posizionale che consente al compilatore di scegliere liberamente i registri generali da usare. Le operazioni che la macro compie, descritte usando assembly standard, sono le seguenti.

1 – Salva i valori di `prev` e `next` nei registri `eax` e `edx` rispettivamente:

```
movl prev, %eax
movl next, %edx
```

2 – Salva il contenuto dei registri `eflags` e `ebp` nello stack in modalità del kernel. Vanno salvati perché il compilatore ipotizza che restino inalterati fino alla fine di `switch_to`:

```
pushfl
pushl %ebp
```

3 – Salva `esp` in `prev->thread.esp`; il campo ora punta alla cima dello stack in modalità del kernel di `prev`

```
movl %esp, 484(%eax)
```

L'operatore identifica la cella di memoria il cui indirizzo è dato dal contenuto di `eax + 484`.

4 – Carica `next->thread.esp` in `esp`. Da ora in avanti, il kernel opera sullo stack in modalità del kernel di `next`. Poiché l'indirizzo del PD è strettamente correlato con lo stack in modalità del kernel, cambiare lo stack significa cambiare il processo corrente:

```
movl 484(%edx), %esp
```

5 – Salva l'indirizzo etichettato 1 in `prev->thread.eip`. Quando il processo sostituito riprende il controllo, esegue l'istruzione etichettata 1

```
movl $1f, 480(%eax)
```

6 – Sullo stack in modalità del kernel di `next` la macro inserisce il valore di `next->thread.eip` che, nella maggior parte dei casi, è l'indirizzo etichettato 1:

```
pushl 480(%edx)
```

7 – Salta alla funzione `__switch_to()`:

```
jmp __switch_to
```

8 – Qui il processo A riprende il controllo della CPU ancora per eseguire alcune istruzioni che ripristinano il contenuto di `eflags` e `ebp`. La prima delle due istruzioni è etichettata 1:

```
1:
    popl %ebp
    popfl
```

Le istruzioni `pop` si riferiscono allo stack in modalità del kernel di `prev`. Queste istruzioni vengono eseguite quando lo scheduler seleziona `prev` come il nuovo processo da eseguire.

9 – Copia il contenuto di `eax` nella locazione di memoria identificata dal terzo parametro `last` della `switch_to`:

```
movl %eax, last
```

Come visto prima, `eax` punta al descrittore del processo appena rimpiazzato³.

La funzione `__switch_to()`

Compie la maggior parte della commutazione di processo. Agisce sui parametri `prev_p` e `next_p`. La chiamata di funzione è diversa dal solito perché essa ricava i parametri `prev_p` e `next_p` dai registri `eax` e `edx` e non dallo stack come di solito. Per forzare la funzione a cercare i parametri nei registri, il kernel usa gli attributi `__attribute__` e `regparm`, che sono estensioni non standard del C implementate dal compilatore `gcc`. La funzione `__switch_to()` è dichiarata nel file `/arch/i386/kernel/process.c`:

```
__switch_to(struct task_struct *prev_p, struct task_struct *next_p) {
    __attribute__((regparm(3)));
```

Le azioni svolte sono le seguenti:

1 – Esegue il codice contenuto nella macro `__unlazy_fpu()` per salvare se necessario il contenuto dei registri FPU, MMX, e XMM del processo `prev_p`.

```
__unlazy_fpu(prev_p);
```

2 – Esegue la macro `smp_processor_id()` per ottenere l'indice della CPU locale, cioè quella che esegue il codice. Lo ottiene dal campo `cpu` della struttura `thread_info` del processo corrente e lo registra nella variabile locale `cpu`.

3 – Carica `next->thread.esp0` nel campo `esp0` del TSS relativo alla CPU locale. Ogni futuro cambiamento del livello di privilegio da modalità utente a modalità del kernel causata dalla istruzione `sysenter` copia questo indirizzo nel registro `esp`:

```
init_tss[cpu].esp0 = next_p->thread.esp0;
```

4 – Carica nella GDT della CPU locale i segmenti Thread-Local Storage (TLS) usato dal processo `next_p`; i tre Selettori di Segmento sono memorizzati nell'array `tls_array` del PD.

```
cpu_gdt_table[cpu][6] = next_p->thread.tls_array[0];
cpu_gdt_table[cpu][7] = next_p->thread.tls_array[1];
cpu_gdt_table[cpu][8] = next_p->thread.tls_array[2];
```

5 – Memorizza il contenuto dei registri di segmento `fs` e `gs` in `prev_p->thread.fs` e `prev_p->thread.gs`:

```
movl %fs, 40(%esi)
movl %gs, 44(%esi)
```

Il registro `esi` punta alla struttura `prev_p->thread`.

6 – Se i registri `fs` e `gs` sono stati usati dai processi `prev_p` o `next_p` (cioè hanno valore diverso da zero), carica in questi registri i valori contenuti nel descrittore `thread_struct` del processo `next_p`. Questa fase completa logicamente le azioni del passo precedente. Le istruzioni sono:

```
movl 40(%ebx), %fs
movl 44(%ebx), %gs
```

³ Nell'implementazione corrente, `schedule()` usa di nuovo la variabile `prev` per cui l'istruzione appare come `movl %eax, prev`.

Il registro ebx punta alla struttura next_p->thread. Il codice è più complesso, dato che potrebbe essere generata una eccezione dalla CPU quando trova un valore di segmento non valido nel registro. Il codice tiene conto di questa possibilità adottando un approccio “fix-up”.

7 – Carica sei registri per il debug, da dr0 a dr7 con il contenuto dell'array next_p->thread.debugreg. Ciò avviene solo se next_p stava usando i registri di debug quando è stato sospeso (cioè quando il campo next_p->thread.debugreg[7] è diverso da 0). Questi registri non hanno bisogno di essere salvati, poiché l'array prev_p->thread.debugreg è modificato solo quando un debugger monitora prev:

```
if(next_p->thread.debugreg[7]) {
    loaddebug(&next_p->thread, 0);
    loaddebug(&next_p->thread, 1);
    loaddebug(&next_p->thread, 2);
    loaddebug(&next_p->thread, 3);
    /* no 4 e 5 */
    loaddebug(&next_p->thread, 6);
    loaddebug(&next_p->thread, 7);
}
```

8 – Aggiorna, se necessario, la bitmap di I/O nel TSS. Questa operazione va compiuta se next_p o prev_p hanno la propria bitmap dei permessi di I/O

```
if(prev_p->thread.io_bitmap_ptr || next_p->thread.io_bitmap_ptr)
    handle_io_bitmap(&next_p->thread, &init_tss[cpu]);
```

Poiché raramente i processi modificano la bitmap, essa viene gestita in modalità “lazy”: la bitmap attuale viene copiata nel TSS della CPU locale solo se un processo accede ad una porta di I/O nello slice di tempo attuale. La bitmap personalizzata di un processo è contenuta in un buffer puntato dal campo io_bitmap_ptr della struttura thread_info. La funzione handle_io_bitmap() imposta il campo io_bitmap del TSS usato dalla CPU locale per il processo next_p:

- se il processo next_p non ha la propria bitmap, il campo io_bitmap del TSS è impostato al valore 0x8000
- se next_p ha la propria bitmap, il campo io_bitmap del TSS è impostato al valore 0x9000.

Il campo io_bitmap dovrebbe contenere un offset entro il TSS dove è contenuta la bitmap. I valori 0x8000 e 0x9000 puntano fuori dal limite di TSS e provocano perciò una eccezione di “General protection” quando il processo in modalità utente tenta di accedere ad una porta di I/O. Il gestore della eccezione do_general_protection() controlla il valore contenuto in io_bitmap: se è 0x8000, invia un segnale SIGSEGV al processo; se è 0x9000 copia la bitmap del processo (puntata dal campo io_bitmap_ptr nella struttura thread_info) nel TSS della CPU locale, imposta il campo io_bitmap all'offset attuale (104), e forza una nuova esecuzione della istruzione assembly.

9 – Termina. La funzione C __switch_to() finisce con l'istruzione

```
return prev_p;
```

Le istruzioni assembly corrispondenti generate dal compilatore sono:

```
movl %edi, %eax
ret
```

Il parametro prev_p (ora in edi) è copiata in eax, perché di default il valore di ritorno di una funzione C è

copiato nel registro `eax`. Da notare che il valore di `eax` è salvato attraverso l'invocazione di `__switch_to()`; questo è importante perché invocare la macro `switch_to` richiede che `eax` contenga sempre l'indirizzo del PD del processo che viene sostituito.

L'istruzione assembly `ret` carica il registro contatore di programma `eip` con l'indirizzo di ritorno inserito in cima allo stack. Comunque la funzione `__switch_to` viene chiamata semplicemente con un salto verso di essa. Perciò l'istruzione `ret` trova sullo stack l'indirizzo della istruzione all'etichetta 1, che è stata inserita dalla macro `switch_to`. Se `next_p` non era mai stato sospeso in quanto eseguito per la prima volta, la funzione trova l'indirizzo di partenza della funzione `ret_from_fork()`.

Salvataggio e caricamento dei registri FPU, MMX e XMM

A partire dal processore 486DX l'unità aritmetica a virgola mobile è stata integrata nella CPU. Continua a venire usato il termine coprocessore matematico, a ricordo dei tempi in cui veniva usato un costoso chip specializzato. Per mantenere la compatibilità con i modelli vecchi, le istruzioni in virgola mobile sono eseguite con istruzioni ESCAPE, che possiedono un prefisso che va da `0xd8` a `0xdf`. Queste istruzioni agiscono sui registri a virgola mobile della CPU. Chiaramente se un processo usa istruzioni ESCAPE, il contenuto dei registri a virgola mobile fa parte del contesto hardware e va salvato.

A partire dal Pentium Intel ha introdotto un nuovo insieme di istruzioni assembly chiamate istruzioni MMX; il loro obiettivo è di velocizzare l'esecuzione delle applicazioni multimediali; esse operano sui registri a virgola mobile. Lo svantaggio è che i programmatori non possono mescolare istruzioni a virgola mobile e MMX. Il vantaggio è che i progettisti del sistema operativo possono ignorare le nuove istruzioni, perché per salvare lo stato dei registri MMX si può sfruttare il codice di commutazione dei task della unità a virgola mobile.

Le istruzioni MMX velocizzano le applicazioni multimediali perché introducono una pipeline a istruzione singola e dati multipli (SIMD). Il Pentium III ne estende la capacità con le estensioni SSE (streaming SIMD Extensions) che aggiungono servizi di gestione dei valori in virgola mobile contenuti negli 8 registri MMX a 128 bit, in modo che istruzioni SSE e FPU/MMX possono essere mescolate liberamente. Il Pentium 4 introduce un'altra caratteristica: le estensioni SSE, che sono fondamentalmente un supporto a valori in virgola mobile con alta precisione. SSE2 usa lo stesso set di registri di XMM come SSE.

I microprocessori 80x86 non salvano automaticamente nel TSS i registri FPU, MMX e XMM. Però hanno un supporto hardware che abilita il kernel a salvarli solo quando necessario. L'hardware consiste nel flag TS (Task-Switching) del registro `cr0`, che segue le regole qui descritte:

- ogni volta che viene compiuta una commutazione hardware, il flag TS è attivato;
- ogni volta che una istruzione ESCAPE, MMX, SSE o SSE2 viene eseguita con il flag TS attivo, si ha un'eccezione di "Device not available" (dispositivo non disponibile)

Un processo A sta usando il coprocessore matematico. Quando avviene una commutazione di contesto tra A e B, il kernel attiva il flag TS e salva i registri in virgola mobile nel TSS di A. Se B non usa il coprocessore matematico, il kernel non deve ripristinare il contenuto dei registri in virgola mobile. Appena B tenta di eseguire una istruzione ESCAPE o MMX, la CPU provoca una eccezione "Device not available" e il gestore corrispondente carica i registri in questione con il valore salvato nel TSS di B.

Le strutture di gestione del caricamento selettivo dei registri FPU, MMX e XMM sono contenute nel sotto-campo `thread.i387` del PD, il cui formato è descritto dalla union `i387_union`:

```
union i387_union {
```

```

        struct i387_fsave_struct    fsave;
        struct i387_fxsave_struct   fxsave;
        struct i387_soft_struct     soft;
};

```

Il campo contiene solo uno dei tre tipi di strutture. Il tipo `i387_soft_struct` è usato dai modelli di CPU non dotati di coprocessore matematico; Linux supporta ancora questi vecchi chip emulando via software il coprocessore. Il tipo `i387_fsave_struct` con coprocessore una unità MMX. Infine `i387_fxsave_struct` viene usato dai modelli di CPU con estensioni SSE e SSE2.

Il PD include due flag aggiuntivi:

- `TS_USED_FPU`, incluso nel campo `status` del descrittore `thread_info`. Indica se il processo ha usato i registri FPU, SSE e MMX.
- `PF_USED_MATH`, incluso nel campo `flags` del descrittore `task_struct`. Specifica se il contenuto del sotto-campo `thread.i387` è significativo. Il flag è azzerato (non significativo) in questi due casi:
 - a - quando il processo esegue un nuovo programma chiamando `execve()`. Poiché il controllo non ritorna più al programma chiamante, i dati del sotto-campo non vengono più usati.
 - b - quando un processo in modalità utente inizia ad eseguire una procedura di gestione di segnale. Poiché il gestore di segnali è asincrono rispetto al flusso di esecuzione del programma, i registri in virgola mobile potrebbero essere privi di significato per esso. Comunque il kernel salva i registri in `thread.i387` prima di avviare il gestore e li ripristina dopo la sua terminazione. Perciò il gestore può usare il coprocessore matematico.

Salvataggio dei registri FPU

Come visto in precedenza, la macro `__switch_to()` esegue la macro `__unlazy_fpu`, passando come argomento il PD di `prev`. La macro controlla il valore dei flags `TS_USED_FPU`. Se è attivato, `prev` ha usato istruzioni FPU, MMX, SSE o SSE2; perciò il kernel deve salvare il relativo contesto hardware:

```

if(prev->thread_info->status & TS_USED_FPU)
    save_init_fpu(prev);

```

La funzione `save_init_fpu()` esegue le seguenti operazioni:

1 – salva il contenuto dei registri FPU nel PD di `prev`, poi reinizializza FPU. Se la CPU usa estensioni SSE/SSE2, salva anche i registri XMM e reinizializza l'unità SSE/SSE2. Un paio di istruzioni assembly inline estese si occupano di tutto:

```

asm volatile( "fxsave %0 ; fnclex": "=m" (prev->thread.i387.fxsave) );

```

se la CPU usa SSE/SSE2, altrimenti:

```

asm volatile( "fnsave %0 ; fwait": "=m" (prev->thread.i387.fsave) );

```

2 – resetta il flag `TS_USED_FPU` di `prev`:

```

prev->thread_info->status &= ~TS_USED_FPU;

```

3 – attiva il flag `CW` del registro `cr0` per mezzo della macro `stts()`, che in pratica contiene le seguenti

istruzioni:

```
movl %cr0, %eax
orl $8, %eax
movl %eax, %cr0
```

Caricamento dei registri FPU

Il contenuto dei registri in virgola mobile non è ripristinato appena il processo riprende l'esecuzione. Comunque il flag TS è stato attivato da `__unlazy_fpu()`. La prima volta che il processo next tenta di eseguire una istruzione ESCAPE, MMX o SSE/SSE2, l'unità di controllo provoca un'eccezione "Device not available" e il kernel (più precisamente il gestore di eccezione coinvolto) esegue la funzione `math_state_restore()`. Il processo next è identificato come current dal gestore.

```
Void math_state_restore() {
    asm volatile ("clts"); /* clear the TS flag of cr0 */
    if(!(current->flags & PF_USED_MATH))
        init_fpu(current);
    restore_fpu(current);
    current->thread.status |= TS_USED_FPU;
}
```

La funzione azzerava i flags CW di cr0, in modo che successive istruzioni FPU, MMX e SSE/SSE2 non provocino l'eccezione "Device not available". Se il contenuto del sotto-campo `thread.i387` non è significativo, `init_fpu()` lo resetta e imposta a 1 il flag `PF_USED_MATH` di current. La funzione `restore` poi carica i registri FPU con i valori contenuti in `thread.i387`. Per fare ciò, è usata l'istruzione `fxrstor` oppure la `frstor`, a seconda se la CPU supporta o no SSE/SSE2. Infine viene settato il flag `TS_USED_FPU`.

Uso delle unità FPU, MMX e SSE/SSE2 in modalità del kernel

Anche il kernel può usare queste unità. Nel farlo, deve evitare di interferire con il processo current in modalità utente. Perciò:

- prima di usare il coprocessore, il kernel chiama `kernel_fpu_begin()` che chiama a sua volta `save_init_fpu()` per salvare i registri se il processo in modalità utente ha usato la FPU, poi resetta il flag TS di cr0.
- Dopo aver usato il coprocessore, il kernel deve chiamare `kernel_fpu_end()` che setta il flag TS di cr0.

In seguito, quando il processo in modalità utente usa il coprocessore, `math_state_restore()` imposta il valore dei registri come in una semplice commutazione di contesto.

Va notato comunque che il tempo di esecuzione di `kernel_fpu_begin()` è piuttosto lungo, tanto da annullare l'aumento di velocità ottenuto con le unità FPU, MMX, SSE/SSE2. Per questo motivo il kernel le usa solo in rari casi, di solito quando deve spostare o cancellare vaste zone di memoria o quando calcola valori di checksum.

Creazione dei processi

I sistemi Unix si basano sulla creazione di processi per soddisfare le richieste degli utenti. Ad esempio, la

shell crea un nuovo processo che esegue un'altra copia della shell ogni volta che l'utente digita un comando.

I sistemi Unix tradizionali trattano tutti i processi allo stesso modo: le risorse del genitore sono duplicate per il figlio. Questo approccio rende la creazione di nuovi processi lenta ed inefficiente, perché richiede la copia dell'intero spazio di indirizzamento del genitore. Raramente il figlio richiede di impiegare tutte le risorse ereditate dal genitore; spesso esegue subito una `execve()` e ripulisce lo spazio di indirizzamento così attentamente copiato.

I sistemi moderni risolvono il problema con tre meccanismi:

- la tecnica Copy On Write (COW) permette sia al figlio che al genitore di leggere le stesse pagine fisiche. Quando uno dei due tenta un'operazione di scrittura, il kernel copia il contenuto in una nuova pagina fisica che viene assegnata al processo scrivente.
- I "lightweight process" (processi "leggeri" - LP) permettono a genitori e figli di condividere molte strutture per-processo del kernel, ad esempio le tabelle di pagina (e quindi l'intero spazio di indirizzamento in modalità utente), le tabelle dei file aperti, i segnali.
- La chiamata di sistema `vfork()` crea un processo che condivide lo spazio di memoria del genitore. Per prevenire sovrascritture di dati necessari al figlio, il genitore è bloccato fino a che il figlio termina o esegue un nuovo programma.

Le chiamate di sistema `clone()`, `fork()` e `vfork()`

I LP sono creati da Linux con la funzione `clone()` che usa i seguenti parametri:

`fn`: indica una funzione che deve essere eseguita dal nuovo processo; quando ritorna, il figlio termina. La funzione restituisce un intero che rappresenta il codice di uscita per il figlio;

`arg`: puntatore ai dati passati alla `fn()`;

`flags`: informazioni varie. Il byte inferiore specifica il numero del segnale da inviare al genitore quando il figlio termina: di solito si tratta di `SIGCHLD`. I restanti tre byte codificano un gruppo di flag di clone elencati successivamente;

`child-stack`: lo stack in modalità utente da assegnare al figlio. Il genitore deve sempre allocare un nuovo stack per il figlio;

`tls`: indirizzo della struttura che definisce un segmento Thread Local Storage per un nuovo LP. Ha significato solo se il flag `CLONE_SETTLS` è attivato;

`ptid`: è l'indirizzo di una variabile del genitore che conterrà il PID del nuovo processo. Ha significato solo se il flag `CLONE_PARENT_SETTID` è attivato;

`ctid`: è l'indirizzo di una variabile del figlio che conterrà il proprio PID. Ha significato solo se il flag `CLONE_PARENT_SETTID` è attivato.

Flags relativi a `clone()`

nome del flag	descrizione
CLONE_VM	Condivide il descrittore di memoria e tutte le Tabelle di Pagina.
CLONE_FS	Condivide la directory root, la directory di lavoro corrente e umask
CLONE_FILES	Condivide la tabella dei file aperti
CLONE_SIGHAND	Condivide la tabella dei gestori di segnale, dei segnali bloccati e pendenti. Se è attivo deve esserlo anche CLONE_VM
CLONE_PTRACE	Se tracciato da un debugger, il genitore vuole che lo sia anche il figlio. Inoltre il debugger può voler tracciare il figlio per conto suo; in questo caso il kernel imposta il flag a 1
CLONE_VFORK	Attivato se è chiamata la vfork()
CLONE_PARENT	Imposta i campi parent e real_parent del PD al valore del genitore del processo chiamante.
CLONE_THREAD	Inserisce il figlio nel thread group del genitore e lo forza a condividere con lui il descrittore di segnale. Vengono impostati di conseguenza il campo tgid e group_leader del figlio. Se è attivato, lo deve essere anche CLONE_SIGHAND.
CLONE_NEWNS	Attivo se il figlio richiede il proprio namespace, cioè la propria prospettiva dei filesystem montati. Non è attivabile assieme a CLONE_FS.
CLONE_SYSVSEM	Condivide le operazioni sui semafori
CLONE_SETTLS	Crea un nuovo TLS per il processo; il segmento è descritto dalla struttura puntata dal parametro tls
CLONE_PARENT_SETTID	Scrive il PID del figlio nella variabile del genitore puntata da ptid
CLONE_CHILD_CLEARTID	Se attivato il kernel imposta un meccanismo da usare quando il figlio termina o esegue un nuovo programma. In questi casi, il kernel azzerla la variabile puntata da ctid e sveglia tutti i processi in attesa dell'evento
CLONE_DETACHED	Ignorato dal kernel
CLONE_UNTRACED	Viene settato dal kernel per sovrascrivere il valore di CLONE_PTRACE
CLONE_CHILD_SETTID	Scrive il PID del figlio nella variabile del figlio puntata da ctid
CLONE_STOPPED	Forza il figlio a partire nello stato TASK_STOPPED

Clone() attualmente è un wrapper definito nella libreria C che imposta lo stack del LP e fa una chiamata di sistema clone() nascosta al programmatore. La routine di servizio sys_clone() che la implementa non ha i parametri fn e arg. Infatti la funzione wrapper salva fn nello stack in corrispondenza del proprio indirizzo di ritorno; arg è salvato subito dopo. Quando la funzione wrapper termina, la CPU recupera l'indirizzo di ritorno ed esegue fn(arg).

La chiamata di sistema fork() è implementata da Linux con clone() in cui il parametro flags specifica che sia il segnale SIGCHLD che tutti i flag clone devono essere azzerati, e nella quale il parametro child_stack è il puntatore allo stack del genitore. Perciò genitore e figlio condividono temporaneamente lo stesso stack in modalità utente. Grazie al meccanismo Copy On Write, ottengono copie separate dello stack appena uno di essi tenta di modificarlo.

La chiamata di sistema vfork() è implementata con clone() in cui il parametro flags specifica che il segnale SIGCHLD e i flag CLONE_VM e CLONE_VFORK devono essere attivi, e il parametro child_stack è il puntatore allo stack corrente del genitore.

La funzione do_fork()

La funzione `do_fork()`, che gestisce le chiamate di sistema `clone()`, `fork()` e `vfork()`, accetta questi parametri:

`clone_flags`: corrisponde al parametro `flags` di `clone()`

`stack_start`: corrisponde al parametro `child_stack` di `clone()`

`regs`: puntatore ai valori dei registri generali salvati nello stack in modalità del kernel quando avviene la commutazione da modalità utente a modalità del kernel

`stack_size`: non usato, vale 0

`parent_tidptr`, `child_tidptr`: corrispondono ai parametri `ptid` e `ctid` di `clone()`.

`do_fork()` fa uso di una funzione ausiliaria chiamata `copy_process()` per impostare il PD e tutte le strutture del kernel richieste per l'esecuzione del processo figlio. Questo in sintesi è ciò che fa:

1 – Alloca un nuovo PID per il figlio cercando nella mappa di bit `pidmap_array`.

2 – Controlla il campo `ptrace` del genitore (`current->ptrace`); se è diverso da 0, il genitore è tracciato da un altro processo, per cui `do_fork()` controlla se il debugger vuole tracciare il figlio a parte (indipendentemente dal valore di `CLONE_PTRACE` di genitore); in questo caso, se il figlio non è un thread del kernel (`CLONE_UNTRACED` azzerato) la funzione setta il flag `CLONE_PTRACE`.

3 – Chiama `copy_process()` per fare una copia del PD. Se tutte le risorse richieste sono disponibili, la funzione restituisce l'indirizzo del descrittore `task_struct` creato. Questo è il nucleo essenziale della procedura di `fork`.

4 – Se il flag `CLONE_STOPPED` è impostato o il processo figlio deve essere tracciato, cioè il flag `PT_PTRACED` è attivato in `p->ptrace`, imposta lo stato del figlio a `TASK_STOPPED` e gli aggiunge un segnale sospeso `SIGSTOP`. Lo stato del figlio rimane nello stato `TASK_STOPPED` finché un altro processo, presumibilmente il processo che lo traccia oppure il genitore, non lo cambia in `TASK_RUNNING`, di solito per mezzo di un segnale `SIGCONT`.

5 – Se il flag `CLONE_STOPPED` non è impostato, chiama `wake_up_new_task()` che esegue le seguenti operazioni:

a - sistema i parametri di scheduling del genitore e del figlio.

b - se il figlio viene eseguito sulla stessa CPU del genitore, ed essi non condividono lo stesso set di tabelle di pagina (`CLONE_VM` azzerato), fa in modo che il figlio sia eseguito prima del padre, inserendolo nella stessa runqueue davanti a lui. Questo accorgimento ha performance migliori se il figlio azzerà il suo spazio di indirizzamento per eseguire un nuovo programma subito dopo la sua creazione. Se si lascia che il genitore venga eseguito prima, il meccanismo Copy On Write comporta una inutile duplicazione di pagine.

c – se il figlio non viene eseguito sulla stessa CPU del padre, o se essi condividono lo stesso set di tabelle di pagina (`CLONE_VM` settato), inserisce il figlio nell'ultima posizione della runqueue del genitore.

6 – Se il flag `CLONE_STOPPED` è attivato, pone il figlio nello stato `TASK_STOPPED`.

7 – Se il genitore è tracciato, memorizza il PID del figlio nel campo `ptrace_message` di `current` e chiama

`ptrace_notify()` che essenzialmente arresta il processo corrente e invia un segnale `SIGCHLD` al genitore. Il "nonno" è il debugger; il segnale `SIGCHLD` notifica al debugger che `current` ha generato un figlio il cui PID può essere trovato in `current->ptrace_message`.

8 – Se è settato il flag `CLONE_VFORK`, inserisce il genitore in una `WQ` e lo sospende fino a che il figlio rilascia il suo spazio di indirizzamento (praticamente quando il figlio termina o esegue un altro programma).

9 – Termina restituendo il PID del figlio.

La funzione `copy_process()`

La funzione `copy_process()` imposta il PD e le altre strutture del kernel richieste per l'esecuzione del processo figlio. I parametri sono gli stessi della `do_fork()`, più il PID del figlio. Essa compie le azioni seguenti:

1 – Controlla se i flag passati nell'argomento `clone_flags` sono compatibili. In particolare restituisce un codice di errore nei seguenti casi:

a – sono settati sia `CLONE_NEWNS` che `CLONE_FS`.

b – `CLONE_THREAD` è attivato ma `CLONE_SIGHAND` è azzerato (i LP nello stesso gruppo devono condividere i segnali)

c – `CLONE_SIGHAND` è attivato ma `CLONE_VM` è azzerato (i LP nello stesso gruppo devono anche condividere il descrittore di memoria).

2 – Esegue controlli di sicurezza chiamando `security_task_create()` prima, poi `security_task_alloc()`. Il kernel 2.6 offre punti di aggancio per estensioni di sicurezza che implementano un sistema più robusto di quello dei tradizionali Unix.

3 – Chiama `dup_task_struct()` per ottenere il PD per il figlio. Questa funzione svolge i compiti seguenti:

a – chiama `__unlazy_fpu()` sul processo corrente per salvare, se necessario, i registri FPU, MMX e SSE/SSE2 nella struttura `thread_info` del genitore. In seguito `dup_task_struct()` copia questi valori nella struttura `thread_info` del figlio;

b - esegue `alloc_task_struct()` per ottenere un PD per il nuovo processo, e memorizza il suo indirizzo nella variabile locale `tsk`;

c – esegue `alloc_thread_info` per ottenere un'area di memoria libera in cui allocare la struttura `thread_info` e lo stack in modalità del kernel del nuovo processo, e salva il suo indirizzo nella variabile locale `ti`. La dimensione di quest'area è 8 o 4 KB;

d – copia il contenuto del descrittore di `current` nella struttura `task_struct` puntata da `tsk`, poi imposta `tsk->thread_info` a `ti`;

e – copia il contenuto di `thread_info` di `current` nella struttura puntata da `ti`, poi imposta `ti->task` a `tsk`;

f – fissa il contatore d'uso del nuovo PD (`tsk->usage`) a 2 per indicare che il PD è in uso e che il processo corrispondente è vivo;

g – restituisce il puntatore al PD del nuovo processo (`tsk`).

4 – Controlla che il valore contenuto in `current->signal->rlim[RLIMIT_NPROC].rlim_cur` sia inferiore o uguale al numero di processi di cui è proprietario l'utente. Se è così, viene restituito un codice di errore, a meno

che il processo non abbia privilegi di root. La funzione ottiene il numero di processi posseduti dall'utente da una struttura chiamata `user_struct`. Questa struttura può essere trovata anche attraverso un puntatore del campo `user` nel PD.

5 – Incrementa il contatore d'uso della `user_struct` (`tsk->user->__count`) e il contatore dei processi posseduti dall'utente (`tsk->user->processes`).

6 – Controlla che il numero dei processi nel sistema (contenuto nella variabile `nr_threads`) non superi il valore della variabile `max_threads`. Il numero di default dipende dalla quantità di RAM del sistema. La regola generale è che lo spazio occupato dai descrittori `thread_info` e `stack` in modalità del kernel non deve superare 1/8 della memoria fisica. L'amministratore può cambiare questo valore scrivendo nel file `/proc/sys/kernel/threads_max`.

7 – Se le funzioni del kernel che implementano il dominio di esecuzione e il formato eseguibile del nuovo processo sono incluse in un modulo del kernel, aumenta il loro contatore d'uso.

8 – Fissa alcuni campi relativi allo stato del processo:

a – inizializza il contatore del “big kernel lock” `tsk->lock_depth` a -1;

b – inizializza `tsk->did_exec` a 0: questo campo tiene il conto del numero di chiamate di sistema `execve()` fatte dal processo;

c – aggiorna alcuni flag inclusi in `tsk->flags` che sono stati copiati del genitore: prima di tutto azzerare `PF_SUPERPRIV`, che indica se il processo ha usato qualche privilegio di superuser, poi imposta `PF_FORKNOEXEC` che indica che il figlio non ha ancora eseguito una chiamata di sistema `execve()`;

9 – Memorizza il PID del nuovo processo nel campo `tsk->pid`.

10 – Se il flag `CLONE_PARENT_SETTID` nel parametro `clone_flags` è attivo, copia il PID del figlio nella variabile indirizzata da `parent_tidptr`.

11 – Inizializza le strutture `list_head` e gli spinlock inclusi nel PD del figlio e imposta altri campi relativi a segnali sospesi, timer, e statistiche.

12 – Chiama `copy_semundo()`, `copy_files()`, `copy_fs()`, `copy_sighand()`, `copy_signal()`, `copy_mm()`, `copy_namespace()` per creare nuove strutture e copiare in esse il valore delle corrispondenti strutture del genitore, a meno che non sia specificato diversamente dal parametro `clone_flags`.

13 – Chiama `copy_thread()` per inizializzare lo stack in modalità del kernel del figlio con i valori contenuti nei registri nel momento in cui `clone()` è stata chiamata (valori salvati nello stack del genitore). Comunque la funzione imposta il valore 0 nel campo corrispondente al registro `eax` (è il valore di ritorno per il figlio della `fork()` o `clone()`). Il campo `thread.esp` nel descrittore del figlio è inizializzato con l'indirizzo della base dello stack in modalità del kernel del figlio, e l'indirizzo di una funzione assembly (`ret_from_fork()`) è memorizzato nel campo `thread.eip`. Se il genitore usa la bitmap dei permessi di I/O, il figlio ne ottiene una copia. Infine se `CLONE_SETTLS` è attivato, il figlio ottiene il segmento TLS specificato dalla struttura puntata dal parametro `tls` di `clone()`⁴.

14 – Se sono attivati `CLONE_SETTID` oppure `CLONE_CHILD_CLEARTID` nel parametro `clone_flags`, la funzione copia il valore di `child_tidptr` in `tsk->set_child_tid` oppure in `tsk->clear_child_tid` rispettivamente. Questi flag specificano che il valore della variabile puntata da `child_tidptr` nello spazio di indirizzamento in modalità utente del figlio deve essere modificato, altrimenti le attuali operazioni di scrittura verrebbero eseguite più tardi.

⁴ Ci si può chiedere come la funzione ottenga il valore di `tls`, dato che non viene passato come parametro alla `do_fork()`; esso viene copiato assieme ad altri nello stack in modalità del kernel. La `copy_thread()` perciò lo cerca nella locazione dello stack corrispondente al valore di `esi`.

15 – Azzerà il flag `TIF_SYSCALL_TRACE` nella struttura `thread_info` del figlio, in modo che la `ret_from_fork()` non dia notizia al debugger della conclusione della funzione. (La chiamata di sistema che traccia il figlio non è disabilitata, poiché è controllata da `PTRACE_SYSCALL` in `tsk->ptrace`).

16 – Inizializza `tsk->exit_signal` col numero di segnale codificato nel bit inferiore del parametro `clone_flags`, a meno che non sia settato `CLONE_THREAD`, nel qual caso inizializza il campo a -1. Solo la fine dell'ultimo componente di un gruppo di thread (di solito il leader) fa partire un segnale per il genitore.

17 – Chiama `sched_fork()` per completare l'inizializzazione delle strutture dello scheduler per il nuovo processo. La funzione imposta anche lo stato del nuovo processo a `TASK_RUNNING` e il campo `preempt_count` della struttura `thread_info` a 1, disabilitando così il pre-rilascio del kernel. Inoltre, per mantenere lo scheduling equilibrato, la funzione divide il tempo di esecuzione rimanente del genitore tra questi e il figlio.

18 – Imposta il campo `cpu` nella struttura `thread_info` con il numero della CPU locale restituito da `smp_processor_id()`.

19 – Inizializza i campi che definiscono le relazioni di parentela. In particolare, se sono attivati `CLONE_PARENT` o `CLONE_THREAD`, inizializza `tsk->real_parent` e `tsk->parent` con `current->real_parent`; in questo modo il genitore del processo corrente è anche il genitore del figlio. Altrimenti imposta i campi a `current`.

20 – Se il figlio non deve essere tracciato (`CLONE_PTRACED` azzerato), imposta a 0 `tsk->ptrace`. Questo campo contiene alcuni flag usati se il processo viene tracciato. In questo modo, anche se è tracciato il genitore, il figlio non lo è.

21 – Esegue la macro `SET_LINKS` per inserire il nuovo PD nella lista dei processi.

22 – Se il figlio deve essere tracciato (`CLONE_PTRACED` settato), imposta `tsk->parent` a `current->parent` e inserisce il figlio nella lista del debugger.

23 – Chiama `attach_pid()` per inserire il PID del nuovo PD nella tabella hash `pidhash[PIDTYPE_PID]`.

24 – Se il figlio è leader di un gruppo di thread (`CLONE_THREAD` azzerato):

a – inizializza `tsk->tgid` a `tsk->pid`;

b – inizializza `tsk->group_leader` a `tsk`;

c – chiama tre volte `attach_pid()` per inserire il figlio nella tabella hash dei PID di tipo `PIDTYPE_TGID`, `PIDTYPE_PGID`, `PIDTYPE_SID`.

25 – Se invece il figlio appartiene al gruppo di thread del genitore (`CLONE_THREAD` attivato):

a – inizializza `tsk->tgid` a `tsk->current->tgid`;

b – inizializza `tsk->group_leader` al valore in `current->group_leader`;

c – chiama `attach_pid()` per inserire il figlio nella tabella hash `PIDTYPE_TGID` (più specificamente nella lista per PID di `current->group_leader`).

26 – Un nuovo processo ora è stato aggiunto: incrementa il valore della variabile `nr_threads`.

27 – Incrementa la variabile `total_forks` per tenere traccia del numero di processi creati tramite `fork`.

28 – Termina restituendo il puntatore al descrittore del figlio (`tsk`).

Dopo che `fork()` è terminata, esiste un processo figlio pronto ad essere eseguito; non è però in esecuzione. Tocca allo scheduler decidere quando potrà ottenere la disponibilità della CPU: in occasione di una commutazione di processo esso concede questa opportunità al processo figlio caricando alcuni registri della CPU con i valori del campo `thread` del suo PD. In particolare in `esp` è caricato il valore di `thread.esp`, e in `eip` il valore di ritorno di `ret_from_fork()`. Questa funzione assembly chiama `schedule_tail()` (che a sua volta chiama `finish_task_switch()` per completare l'operazione), carica negli altri registri i valori salvati nello stack e riporta la CPU in modalità utente. Il nuovo processo inizia la sua esecuzione subito dopo la fine di `fork()`, `vfork()` o `clone()`. Il valore di ritorno delle chiamate di sistema è contenuto in `eax`: è 0 per il figlio, ed è il PID del figlio per il genitore. Per capire come ciò avviene, basta guardare a come imposta il valore di `eax` la `copy_thread()` al punto 13 di `copy_process()`.

Il figlio esegue lo stesso codice del genitore; la differenza è che `fork` restituisce 0. Lo sviluppatore di applicazioni sfrutta il valore di ritorno nel modo familiare nel mondo Unix, inserendo una espressione condizionale nel programma basata sul valore del PID; in questo modo può forzare il figlio a comportarsi diversamente dal genitore.

Thread del kernel

I sistemi Unix tradizionali delegano a processi che vengono eseguiti ad intermittenza alcune funzioni importanti, come il flush della cache dei dischi, lo swap delle pagine non usate, la gestione delle connessioni di rete, e così via. Non è conveniente eseguire questi compiti in modo strettamente sequenziale; sia le funzioni di gestione che i processi utente finali hanno migliori performance se sono schedulati in background. Poiché alcuni processi sono eseguiti solo in modalità del kernel, i sistemi operativi moderni delegano queste funzioni a *thread del kernel (KT)*, che non sono gravati dal contesto di esecuzione in modalità utente. In Linux i KT differiscono dai processi normali per due caratteristiche:

- sono eseguiti solo in modalità del kernel, mentre i processi si alternano modalità utente e modalità del kernel;
- di conseguenza, usano solo indirizzi lineari superiori a `PAGE_OFFSET`. I processi normali invece usano tutti i 4 GB.

Creazione di un thread del kernel

La funzione `kernel_thread()` crea un nuovo KT; riceve come parametri l'indirizzo della funzione da eseguire (`fn`), l'argomento da passare alla funzione (`arg`) e un set di flag clone (`flags`). Sostanzialmente chiama `do_fork()` in questo modo:

```
do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
```

`CLONE_VM` evita la duplicazione delle tabelle di pagina del processo chiamante: sarebbe uno spreco di memoria perché il KT non usa mai lo spazio di indirizzamento dello modalità utente. `CLONE_UNTRACED` garantisce che nessun processo possa tracciare il KT, anche se il chiamante è tracciato.

Il parametro `pregs` corrisponde all'indirizzo nello stack in modalità del kernel in cui `copy_thread()` trova il valore iniziale dei registri della CPU per il nuovo KT. La funzione `kernel_thread()` imposta lo stack in modo che

- nei registri `ebx` e `edx` vengano copiati da `copy_thread()` i valori dei parametri `fn` e `arg`;
- il registro `eip` punti al seguente frammento di codice:

```
movl %edx, %eax
pushl %edx
call *%ebx
pushl %eax
call do_exit
```

Perciò il KT inizia eseguendo la funzione `fn(arg)`. Se essa termina, il KT esegue `_exit()` passandole il valore di ritorno di `fn()`.

Processo 0

Il progenitore di tutti i processi, chiamato *processo 0* o *processo idle* o, per motivi storici, *swapper*, è un KT creato da zero durante l'inizializzazione di Linux. Esso usa le seguenti strutture allocate staticamente:

- un descrittore di processo memorizzato nella variabile `init_task` inizializzata dalla macro `INIT_TASK`;
- un descrittore `thread_info` e uno stack memorizzati nella variabile `init_thread_union` inizializzata dalla macro `INIT_THREAD_INFO`;
- le seguenti tabelle a cui punta il PD:
 - `init_mm` inizializzata dalla macro `INIT_MM`
 - `init_fs` inizializzata da `INIT_FS`
 - `init_files` inizializzata da `INIT_FILES`
 - `init_signals` inizializzata da `INIT_SIGNALS`
 - `init_sighand` inizializzata da `INIT_SIGHAND`
- la master kernel Page Global Directory memorizzata in `swapper_pg_dir`.

La funzione `start_kernel()` inizializza tutte le strutture necessarie al kernel, abilita gli interrupt e crea un altro KT, chiamato processo 1 (o `init`).

```
kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
```

Il processo `init` ha PID 1 e condivide tutte le strutture del processo 0. Quando viene selezionato dallo scheduler esegue la funzione `init()`.

Dopo aver creato il processo `init`, il processo 0 esegue la funzione `cpu_idle()`, che ripete l'istruzione `hlt` con gli interrupt abilitati. Il processo 0 viene scelto dallo scheduler solo quando non ci sono altri processi nello stato `TASK_RUNNING`.

Nei sistemi multiprocessore c'è un processo 0 per ogni CPU. Subito dopo l'accensione, il BIOS fa partire una sola CPU, mentre le altre sono disabilite. Il processo 0 inizializza le strutture del kernel, poi fa partire le altre CPU e crea i relativi processi `swapper` per mezzo di `copy_process()` passando il valore 0 come nuovo PID. Il kernel imposta il campo `cpu` del descrittore `thread_info` di ogni processo generato col valore appropriato di CPU.

Processo 1

Il KT figlio del processo 0 esegue la funzione `init()` che completa l'inizializzazione del kernel. Poi `init()` chiama `execve()` per caricare il programma eseguibile `init`. Come risultato, il KT `init` diventa un processo normale con le proprie strutture dati per-processo del kernel, e rimane vivo fino allo spegnimento del sistema poiché crea e controlla tutti i processi.

Altri thread del kernel

Linux usa diversi KT; alcuni sono creati in fase di inizializzazione e terminano solo allo spegnimento del sistema; altri sono creati al bisogno quando il kernel deve eseguire un compito che viene svolto meglio nel proprio contesto di esecuzione.

Alcuni esempi di KT:

`keventd` (chiamato anche `events`): esegue le funzioni nella coda `kevent_wq`;

`kapmd`: gestisce Advanced Power Management (APM);

`kswapd`: recupera memoria al bisogno;

`pdflush`: esegue il flush dei buffer "dirty" per recuperare memoria;

`kblockd`: esegue le funzioni nella coda `kblockd_workqueue`. Di fatto attiva periodicamente i driver dei dispositivi a blocchi;

`ksoftirqd`: esegue i tasklets.

Eliminazione dei processi

La maggior parte dei processi "muore" nel senso che termina l'esecuzione del proprio codice. Quando ciò accade, il kernel deve essere avvisato per poter liberare le risorse impegnate dal processo: memoria, file aperti, ed ogni cosa che fa parte del sistema, come i semafori.

Il modo normale di terminare è la chiamando la funzione di libreria `exit()`, che libera le risorse allocate dalla libreria C, esegue ogni funzione indicata dal programmatore e termina con una chiamata di sistema che elimina il processo dal sistema. La funzione di libreria `exit()` può essere chiamata esplicitamente dal programmatore; altrimenti è il compilatore C a inserire la chiamata ad `exit()` dopo l'ultima istruzione di `main()`.

In alternativa, il kernel può forzare la fine di un intero gruppo di thread. Ciò accade quando un processo nel gruppo riceve un segnale che non può gestire o ignorare, o quando viene generata in modalità del kernel un'eccezione non recuperabile mentre il kernel sta operando per conto del processo.

Fine del processo

In Linux esistono due chiamate di sistema che fanno terminare un'applicazione in modalità utente:

- `exit_group()`, che fa terminare un intero gruppo di thread, cioè una applicazione multithread. La funzione che la implementa è chiamata `do_group_exit()`. Questa è la chiamata di sistema utilizzata dalla funzione `exit()`.
- `Exit()`, che fa terminare un singolo processo, ignorando gli altri elementi del gruppo. La funzione che

la implementa è `do_exit()`. E' la chiamata di sistema utilizzata dalla funzione `pthread_exit()` della libreria `LinuxThreads`.

Funzione `do_group_exit()`

Questa funzione uccide tutti i processi che appartengono al gruppo di thread di `current`. Riceve come parametro il codice di uscita, cioè il valore specificato da `exit_group()` (terminazione normale) o un codice di errore fornito dal kernel (terminazione anomala). Esegue le operazioni seguenti:

1 - controlla se il flag `EXIT_GROUP_SIGNAL` del processo che sta terminando non è zero, che significa che il kernel ha già avviato una procedura di `exit`. In questo caso considera come codice di uscita il valore `current->signal->group_exit_code`, e salta al punto 4.

2 - Nel caso contrario, attiva il flag `EXIT_GROUP_SIGNAL` e memorizza il codice di terminazione nel campo `current->signal->group_exit_code`.

3 - Chiama la funzione `zap_other_threads()` per uccidere gli altri processi del gruppo, se esistono. Per fare questo la funzione scandisce la lista per PID della tabella hash `PIDTYPE_TGID`; per ogni processo diverso da `current` invia un segnale `SIGKILL`. Di conseguenza tutti i processi del gruppo eseguono `do_exit()` e terminano.

4 - Chiama `do_exit()` passandole il codice di uscita. `do_exit()` uccide il processo e non ritorna mai.

Funzione `do_exit()`

Tutti i processi di terminazione sono gestite da `do_exit()` che rimuove la maggior parte dei riferimenti al processo che sta terminando dalle strutture del kernel. Essa riceve come parametro il codice di uscita ed esegue le seguenti azioni:

1 - imposta il flag `PF_EXITING` nel campo `flag` del PD ad indicare che il processo sta per essere eliminato.

2 - Rimuove se necessario il PD dalla coda del timer dinamico per mezzo di `del_timer_sync()`.

3 - Distacca dal PD le strutture correlate alla paginazione, ai semafori, al filesystem, ai descrittori di file, ai namespace, alla bitmap dei permessi di I/O per mezzo di `exit_mm()`, `exit_sem()`, `__exit_files()`, `__exit_fs()`, `exit_namespace()`, `exit_thread()`. Queste funzioni rimuovono le strutture se nessun altro processo le condivide.

4 - Se le funzioni del kernel che implementano il dominio e il formato di esecuzione del processo sono incluse in moduli del kernel, la funzione decrementa il loro contatore di uso.

5 - Memorizza nel campo `exit_code` del PD il codice di terminazione del processo. Questo valore è il parametro di `exit()` o `exitgroup()` in caso di terminazione normale, o un codice di errore fornito dal kernel in caso di terminazione anomala.

6 - Chiama `exit_notify()` per compiere le seguenti operazioni:

a - Aggiorna le relazioni di parentela dei figli e del genitore. I figli del processo che sta per terminare diventano figli di un altro processo dello stesso gruppo di thread, se esiste, oppure di `init`.

b - Controlla se il campo `exit_signal` del PD è diverso da 1, e se il processo è l'ultimo membro di un gruppo di thread. In questo caso invia un segnale (di solito `SIGCHLD`) al genitore per informarlo della fine del figlio.

c – Se `exit_signal = -1` o il gruppo include altri processi, invia un `SIGCHLD` al genitore solo se il processo è attualmente tracciato da un debugger (che in questo caso è il genitore).

d – Se `exit_signal = -1` ma il processo non è tracciato, imposta `exit_state` a `EXIT_DEAD` e chiama `release_task()` per liberare la memoria occupata dalle strutture del processo e decrementare il contatore di uso del PD. Il contatore diventa uguale a 1 (vedi punto 3f di `copy_process()`) così che il PD non viene rilasciato subito;

e – Se `exit_signal` è diverso da -1 oppure il processo è tracciato, , imposta `exit_state` a `EXIT_ZOMBIE`.

f – imposta il flag `PF_DEAD` nel campo `flags` del PD.

7 – Chiama la funzione `schedule()` per selezionare un nuovo processo da eseguire. Poiché un processo nello stato di `EXIT_ZOMBIE` viene ignorato dallo scheduler, il processo arresta la sua esecuzione subito dopo che `schedule()` ha chiamato la macro `switch_to`. Lo scheduler controlla il flag `PF_DEAD` e decrementa il contatore d'uso nel PD per indicare che il processo zombie non è più vivo.

Rimozione del processo

Unix consente ad un processo di ottenere dal kernel il PID del genitore o lo stato dell'esecuzione di un figlio. Un processo può, ad esempio, creare un processo figlio per eseguire un compito specifico e poi chiamare una funzione del gruppo `wait()` per verificare se il figlio ha terminato. In questo caso il codice di terminazione informa il genitore se il lavoro è stato portato a termine.

Per soddisfare queste scelte di progetto, il kernel non è autorizzato a scartare i dati contenuti nel PD prima che il processo termini. Lo può fare solo dopo che il genitore ha eseguito una chiamata di sistema del gruppo `wait()`. Questo è il motivo per cui è stato introdotto lo stato `EXIT_ZOMBIE`: anche se tecnicamente il processo è morto, il suo descrittore deve essere conservato fino a che il genitore non venga informato dell'accaduto.

Cosa succede se il genitore termina prima del figlio? Il sistema potrebbe essere invaso da processi zombie i cui descrittori rimarrebbero per sempre in memoria. Il problema viene risolto imponendo che tutti i processi orfani diventino figli di `init`. In questo modo è `init` a distruggere gli zombie controllando la terminazione dei suoi figli per mezzo delle `wait()`.

La funzione `release_task()` distacca le ultime strutture dal PD del processo zombie; viene applicata in due modi possibili: per mezzo della funzione `do_exit()` se il genitore non è interessato a ricevere segnali dal figlio, o delle chiamate di sistema `wait4()` o `waitpid()` dopo che è stato inviato al genitore un segnale. In quest'ultimo caso, la funzione libera anche la memoria usata dal PD, mentre nel primo caso è lo scheduler a liberarla. La funzione esegue le seguenti azioni:

1 – decrementa il numero di processi che appartengono al proprietario del processo terminato. Il valore è contenuto nella struttura `user_struct` già menzionata in precedenza.

2 – Se il processo è tracciato, la funzione lo rimuove dalla lista `ptrace_children` del debugger e lo riassegna al genitore originale.

3 – Chiama `__exit_signal()` per cancellare ogni segnale sospeso e liberare il descrittore `signal_struct`. Se il descrittore non è più usato da altri LP, la funzione rimuove questa struttura dati. Inoltre chiama `exit_itimers()` per distaccare dal processo ogni timer POSIX.

4 – Chiama `__exit_sighand()` per liberarsi del gestore di segnali.

5 – Chiama `__unhash_process()` che esegue le seguenti azioni:

a - decrementa di 1 la variabile `nr_threads`;

b - chiama due volte `detach_pid()` per rimuovere il PID dalle tabelle hash `PIDTYPE_PID` e `PIDTYPE_TGID`;

c – se il processo è il leader di un gruppo chiama ancora `detach_pid()` due volte per rimuovere il PD dalle tabelle hash `PIDTYPE_PGID` e `PIDTYPE_SID`;

d – usa la macro `REMOVE_LINKS` per scollegarlo dalla lista dei processi.

6 – se il processo non è il leader di un gruppo di thread, il leader è uno zombie e il processo è l'ultimo elemento del gruppo, la funzione invia un segnale al genitore per informarlo della morte del processo.

7 – Chiama `sched_exit()` per regolare il time slice del genitore (vedi passo 17 di `copy_process()`).

8 – chiama `put_task_struct()` per decrementare il contatore d'uso; se questo diventa zero, la funzione elimina ogni residuo riferimento al processo:

a – decrementa il contatore d'uso `__count_field` della struttura `user_struct` del proprietario del processo e la rilascia se il valore del contatore diventa 0;

b – rilascia il PD e l'area di memoria usata per il descrittore `thread_info` e lo stack in modalità del kernel.