Laboratorio di Reti e Sistemi Distribuiti

13: Task Management

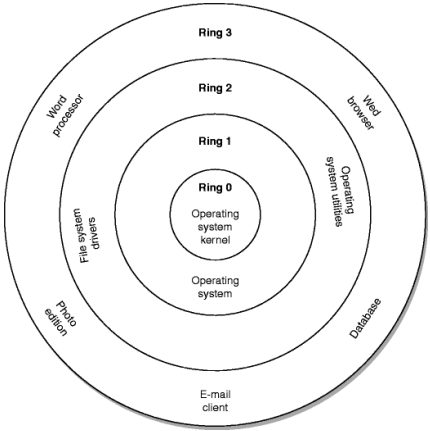
Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 3rd April 2025

Level Protection Rings

I "ring" (anelli) sono livelli di privilegio definiti dall'architettura della CPU per isolare e proteggere le operazioni del sistema operativo e delle applicazioni. Nei sistemi moderni, i più rilevanti sono **Ring 0** (modalità kernel) e **Ring 3** (modalità utente).



Level Protection Rings

I "ring" sono livelli gerarchici di protezione che limitano cosa può fare un processo in esecuzione su una CPU.

- Ring 0 (più privilegiato): Accesso completo all'hardware.
- Ring 3 (meno privilegiato): Accesso limitato, per applicazioni utente.
  Gli anelli intermedi (Ring 1-2) sono raramente usati nei sistemi moderni.

Ring 0

Ring 0 (Modalità Kernel): Usato dal sistema operativo e dai driver dell'hardware.

- Accesso diretto all'hardware (es: gestione della memoria, dispositivi I/O).
- Istruzioni CPU privilegiate (es: modifica della tabella delle pagine, abilitazione degli interrupt).
- Accesso completo alla memoria (kernel space e user space).

Ring 3

Ring 3 (Modalità Utente): Usato dalle applicazioni utente (es: browser, editor di testo).

- Nessun accesso diretto all'hardware (richiede system call).
- Istruzioni CPU limitate (es: non può modificare le impostazioni di virtualizzazione).
- Accesso solo alla memoria user space (isolato dal kernel).

Transizione tra Ring 0-3

- System call: Quando un'applicazione (Ring 3) deve eseguire un'operazione privilegiata (es: aprire un file), usa una system call. Il kernel valida la richiesta ed esegue l'operazione.
- Interrupt hardware: Un dispositivo (es: tastiera) genera un interrupt, spostando l'esecuzione in Ring 0 per gestirlo.
- Overhead: Ogni transizione tra Ring 3 e Ring 0 comporta un costo prestazionale (salvataggio del contesto, cambio di protezione).

Ring -1: Virtualization Ring

Alcune CPU moderne (Xeon) introducono un "anello" ulteriore per gli hypervisor (Ring -1), che gestiscono macchine virtuali con privilegi superiori al kernel

User Stack vs Kernel Stack

User Stack

Lo stack utente (user stack) è una regione di memoria allocata nel contesto di un singolo thread in modalità utente (user space).

- **Contenuto:** Variabili locali, indirizzi di ritorno, parametri di funzione.
- **Gestione:** Creato e gestito automaticamente dal runtime (es: compilatore, libreria C).
- **Dimensione:** Dinamica fino a un limite massimo (es: 8 MB su Linux).
- Ogni thread ha il proprio user stack separato, anche se appartenente allo stesso processo.

User Stack vs Kernel Stack

User Heap

- Lo **heap utente** è una regione di memoria condivisa all'interno dello user space di un processo, utilizzata per allocazioni dinamiche (es: malloc() in C).
- **Contenuto:** Dati allocati dinamicamente durante l'esecuzione.
- **Gestione:** Controllato dall'applicazione (es: malloc, free).
- **Condivisione:** Tutti i thread di un processo condividono lo stesso heap.

User Stack vs Kernel Stack

Kernel Stack

Il **kernel stack** è una regione di memoria riservata nel kernel space, associata a ciascun thread per gestire operazioni in modalità kernel.

- Contenuto: Dati temporanei del kernel, struct thread_info.
- Dimensione: Fissa (es: 8 KB o 16 KB).
- Accesso: Usato solo durante system call, interrupt, o context switch.

Kernel Heap

Esiste un **heap separato nel kernel space** (es: per kmalloc), ma non è accessibile dalle applicazioni user space.

Recap Processi e Threads

- Il processo è l'unita di astrazione fondamentale in Unix. E' definito come un "programma in esecuzione".
- Un processo include un insieme di risorse: file aperti, segnali in attesa, stati del processore, spazio di indirizzamento e soprattutto...
- flussi di esecuzione **(thread)** che sono gli oggetti attivi all'interno del processo
- ogni thread include un unico Progam Counter, Stack, Registri
- Il Kernel non schedula processi ma Thread
- Il thread è un particolare tipo di processo. A basso livello i due concetti vengono a coincidere e vengono denominati TASK

Struct task_struct

È il **descrittore principale di un processo o thread** nel kernel Linux. Contiene tutte le informazioni necessarie per gestire un task, che sia un processo intero o un thread (in Linux, i thread sono implementati come processi leggeri "lightweight processes", LWP).

```
1 struct task_struct {
2     volatile long state;              // Stato del processo
3     pid_t pid;                        // Process ID
4     struct task_struct *parent;       // Processo padre
5     struct list_head children;        // Lista dei processi figli
6     struct mm_struct *mm;             // Gestione della memoria
7     struct files_struct *files;       // File aperti
8     // ... altri campi ...
9 };
```

struct thread_info

Struttura posta alla base dello stack del kernel di ogni thread. Contiene informazioni specifiche del thread e un collegamento diretto al task_struct. Ottimizza l'accesso ai dati critici sfruttando la posizione fissa nello stack del kernel.

```
1 struct thread_info {
2     struct task_struct *task;       // Puntatore al task_struct
3     unsigned long flags;            // Flag (es: TIF_NEED_RESCHED)
4     int preempt_count;              // Contatore per la prelazione
5     mm_segment_t addr_limit;        // Limite degli indirizzi (kernel/
      user)
6     // ... altri campi ...
7 };
```

Current Task Retrieval

- Serve ad ottenere un puntatore alla struttura struct task_struct del task (processo/thread) attualmente in esecuzione sulla CPU.
- E' usata nel codice del kernel per gestire operazioni legate al contesto corrente (es: ottenere il PID, credenziali, stato del processo).

```
1 #define current (current_thread_info()->task)
2
3 static inline struct thread_info *current_thread_info(void) {
4     return (struct thread_info *)(current_stack_pointer & ~(
    THREAD_SIZE - 1));
5 }
```

Process State

In Linux, un processo può trovarsi in diversi stati che ne descrivono il ciclo di vita. Questi stati sono visibili tramite strumenti come `ps` o `top`.

- `R (Running)`: Il processo è in esecuzione sulla CPU o in attesa nella coda di schedulazione (pronto per essere eseguito).
- `S (Interruptible Sleep)`: Il processo attende un evento (es. input utente, I/O). Può essere risvegliato da un segnale.
- `D (Uninterruptible Sleep)`: Stato critico durante operazioni di I/O (es. scrittura su disco). Non può essere interrotto.
- `T (Stopped)`: Processo sospeso manualmente (es. con `Ctrl+Z`) o da un debugger. Si riprende con `SIGCONT`.
- `Z (Zombie)`: Processo terminato ma non rimosso dalla tabella dei processi. Risolto terminando il processo padre.
- `X (Dead)`: Stato transitorio prima della rimozione definitiva del processo.

Process Context ed Interrupt Context

Contesto di processo

Quando un programma/processo chiama una syscall entra in kernel space. Da questo momento si dice che il kernel "esegue per conto del processo" e si trova in contesto di processo. Da questo momento in poi la macro **current** è valida.

Contesto di interrupt

Quando viene **lanciata una interruzione** il kernel passa in contesto di interrupt e non opera per conto del processo in esecuzione. La macro **current** non è valida.

Kernel threads

I **kernel thread** sono thread (flussi di esecuzione) creati e gestiti direttamente dal kernel del sistema operativo. A differenza dei processi utente, operano esclusivamente in spazio kernel e non dispongono di uno spazio di indirizzamento utente (kthread_create()).

- Non eseguono codice utente
- Non hanno un processo genitori
- Sono solitamente thread di servizio
- Stessa coda dei task ma priorità e politiche di scheduling diverse

Esempi

- **kswapd**: Recupera memoria libera "swappando" pagine non utilizzate su disco.
- **kcompactd**: Compatta la memoria per ridurre la frammentazione.

clone()

In Linux, clone() è una system call di basso livello utilizzata per creare nuovi **TASK**, con un controllo preciso su cosa viene condiviso tra il processo padre e il figlio. È più flessibile di fork() (per i processi) e pthread_create() (per i thread), ma richiede una gestione manuale.

```
1 #include <sched.h>
2
3 int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

1. **fn**: Funzione da eseguire nel nuovo processo/thread.
2. **child_stack**: Puntatore allo stack del figlio (deve essere allocato manualmente).
3. **flags**: Flag che determinano cosa viene condiviso (es. memoria, file descriptor).
4. **arg**: Argomento passato alla funzione fn.

Process Creation with clone()

```c
int child_function(void *arg) {
    printf("Sono il figlio (PID: %d)\n", getpid());
    return 0;
}

int main() {
    // Alloca lo stack per il figlio (8 MB)
    void *stack = malloc(8192 * 1024);

    // Crea un processo con stack separato e nessuna condivisione
    int pid = clone(child_function, stack + 8192 * 1024, SIGCHLD, NULL
    );
    free(stack);
    return 0;
}
```

Thread Creation with clone()

```c
int thread_function(void *arg) {
    printf("Sono un thread (PID: %d)\n", getpid());
    return 0;
}

int main() {
    void *stack = malloc(8192 * 1024);

    // Flag per thread: condividi memoria, file descriptor, etc.
    int flags = CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND;

    clone(thread_function, stack + 8192 * 1024, flags, NULL);
    free(stack);
    return 0;
}
```