# Processi in Linux

---

## Index



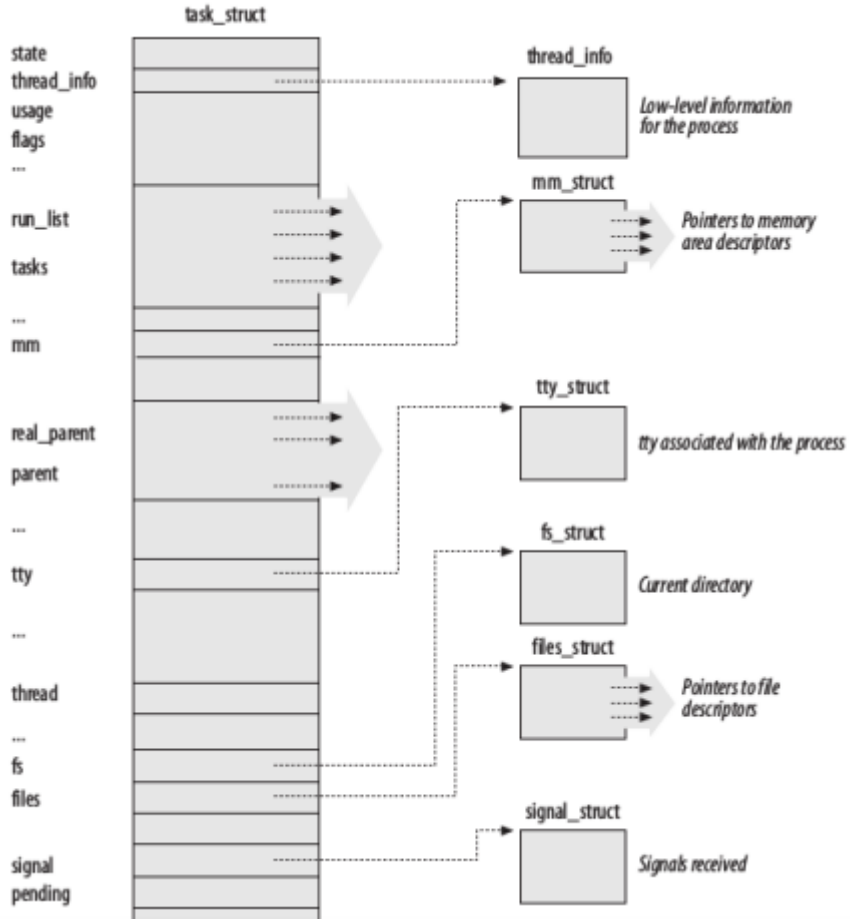
---

## Processi e thread

Derivando da UNIX, che non ha i thread, la loro implementazione all'interno di Linux è stata particolarmente articolata ed è per questo che sono ben diversi da come sono stati mostrati fino ad ora.
In Linux l'unità di base sono i thread (è come se la `fork` creasse il thread), infatti i processi stessi sono chiamati Lightweight process (**LWP**).
In questo SO sono possibili **sia i KLT** (usati principalmente dal sistema operativo) **che gli ULT** (che possono essere direttamente scritti da un utente e che tramite la libreria `pthread` vengono essere poi mappati in KLT)

> ⚠ **Warning**
>
> Il PID è un identificativo unico che vale per tutti i thread dello stesso processo viene quindi introdotto un **`tid`** (task identifier) che identifica ogni singolo thread.
> Come abbiamo detto ogni processo ha almeno un thread associato, questo thread ha il **TID uguale al PID**.
> Dunque è chiamato process identifier ma in realtà è un thread identifier, questo poiché l'unità di base è l'LWP, che coincide con il concetto di thread.
> L'entry del PCB che dà il PID comune a tutti i thread di un processo è il **`tgid`** (thread group identifier), e coincide con il PID del primo thread del processo
> Una chiamata a `getpid()` restituisce il `tgid`
> Ovviamente per processi a singolo thread `tgid` e `pid` coincidono

In Linux inoltre è presente **un PCB per ogni thread**, diversi thread dunque conterranno informazioni duplicate




`thread_info` → è organizzata per contenere anche il kernel stack, ovvero lo stack delle chiamate da usare quando il processo passa in modalità sistema (system call)
`thread_group` → punta agli altri thread dello stesso processo
`parent` e `real_parent` → puntano la padre del processo (ci sono anche link per i fratelli e figli)

---

## Stati dei processi

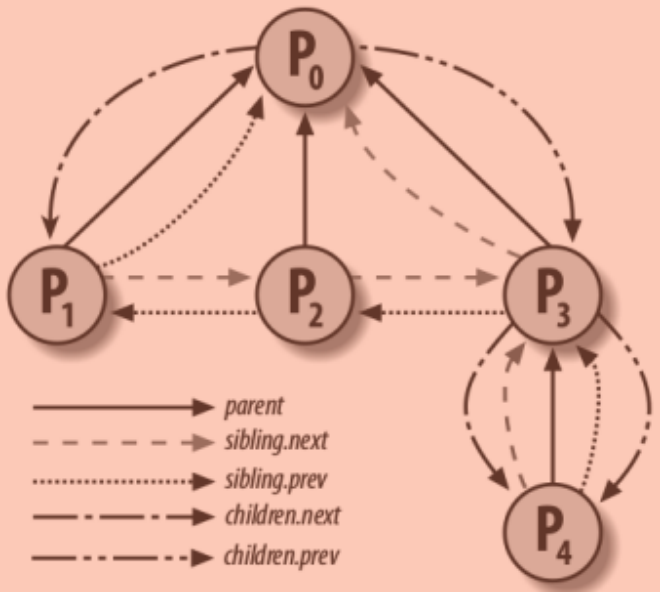
E' sostanzialmente come quello a 5 stati (sono presenti i processi suspended ma non vi è fatta un'esplicita menzione)
Gli stati sono i seguenti:

- `TASK_RUNNING` → include sia Ready che Running (se uno processo è davvero running sta già sul processore non è dunque necessario distinguerli)
- `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_STOPPED`, `TASK_TRACED` → sono tutti Blocked che si differenziano per il motivo per cui sono blocked; in ordine blocked che non presenta problemi, connesso ad alcune operazioni I/O su dischi che sono particolarmente lenti per qualche motivo e non permette alcun tipo di azione, è stato esplicitamente bloccato, sto facendo debugging

- `EXIT_ZOMBIE`, `EXIT_DEAD` → entrambi stati di Exit

---

## Processi parenti




In ogni PCB sono presenti informazioni per risalire ai processi a lui connessi. Si può infatti accedere ai fratelli (processi che hanno lo stesso padre).
In aggiunta a questo ogni processo ha i suoi thread (non rappresentati)

---

## Segnali ed interrupt

Non bisogna confondere i segnali con interrupt (o eccezioni).
I **segnali** infatti possono essere inviati **da un processo utente ad un processo utente** (tramite una system call, chiamata `kill` ma potrebbe accadere che non termini il processo in rari casi).

Quando viene inviato un segnale, questo viene aggiunto all'opportuno campo del PCB del processo ricevente. A questo punto, quando il processo viene nuovamente schedulato per l'esecuzione, il kernel controlla prima se ci sono segnali pendenti; se si esegue un'opportuna funzione chiamata *signal handler* (a differenza dell'interrupt handler, questo viene eseguito in **user mode**). I signal handler possono essere di sistema possono essere sovrascritti da signal handler definiti dal programmatore (alcuni segnali hanno handler non sovrascribili)

I segnali possono anche essere inviati da un processo in modalità sistema, ma in questo caso molto spesso ciò è dovuto ad un interrupt a monte.
Esempio tipico: eseguo un programma C scritto male, che accede ad una zona di

memoria senza averla prima richiesta, il processore fa scattare un'eccezione, viene eseguito l'opportuno exception handler (in kernel mode) che essenzialmente manda il segnale `SIGSEGV` (violazione di segmento, segmentation fault) al processo colpevole, quando il processo colpevole verrà selezionato nuovamente per andare in esecuzione, il kernel vedrà dal PCB che c'è un segnale pendente, e farà in modo che venga eseguita l'azione corrispondente. L'azione di default per tale segnale è di far terminare il processo (fa una system call che ritorna in kernel mode e termina il processo; può essere riscritta dall'utente quando verrà eseguita tale azione, sarà in user mode)

Dunque le differenze fondamentali tra segnali ed interrupt sono:

- i signal handler sono eseguiti in user mode mentre gli interrupt handler in kernel mode
- i signal handler potrebbero essere riscritto dal programmatore mentre gli interrupt handler no