Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N1 – Introduzione a Linux

N.1 Introduzione a LINUX

1 Aspetti generali di Linux

Il SO Linux appartiene alla famiglia dei sistemi Unix, il cui sviluppo è iniziato negli anni 70.
Il primo obiettivo del sistema Linux, il cui sviluppo è iniziato nel 1991, è stato quello di creare un sistema Unix adatto alla gestione di un Personal Computer in architettura Intel 386. Successivamente il SO Linux ha trovato applicazione in altri contesti e attualmente le applicazioni in cui è presente sono le seguenti:

- PC: in questo campo, dominato dal sistema Windows, Linux ha avuto un successo parziale
- Server: questo è il settore di maggior successo di Linux, che costituisce il sistema dominante nella gestione dei Server
- Sistemi embedded: in questo settore l'impiego di Linux è in continua crescita;
- Sistemi Real-Time: Linux in origine non era in grado di supportare i sistemi real-time, ma il suo adattamento per supportare tali sistemi è molto progredito e in continua evoluzione
- Sistemi Mobile: Linux è alla base di Android, uno dei più diffusi sistemi operativi mobile

Come vedremo, l'uso di Linux in diversi settori ha influenzato e continuerà a influenzare numerose scelte progettuali del sistema stesso.
Linux evolve continuamente, quindi è praticamente impossibile fornirne una descrizione allineata alla versione più recente. In queste note si fa riferimento principalmente alla versione 2.6.15 e ad alcune delle successive modificazioni. I programmi presentati sono stati eseguiti tutti su una versione recente (minimo 3.2).
In Tabella 1 è riportata in estrema sintesi la storia delle versioni di Linux.

numero	data	righe codice (in K)	note
0.01	1991	10	prima versione
0.95	1992		X window system
1.0	1994	176	
2.0	1996		
2.2.0	1999	1.800	
2.2.13	1999		inizio uso come macchina server enterprise
2.4.0	2001	3.377	
2.6.0	2003	5.929	
2.6.15	2006		Introduzione scheduler CFS
3.10	2013	15.803	
4.0	2015		

Tabella 1

Alcune osservazioni:

- l'evoluzione quasi esponenziale della quantità di codice di Linux dipende in primo luogo dalla crescita del numero di dispositivi periferici nuovi supportati; questa crescita è continua e inarrestabile a causa dell'evoluzione dell'Hardware – al momento i gestori di periferiche occupano più del 50% del codice complessivo di Linux
- tra la versione 2.0 e la 2.6 si è verificato un aumento della complessità del sistema dovuto a due fenomeni:
    - la sostituzione delle strutture statiche (array) delle prime versioni di Linux con strutture dinamiche, con conseguente eliminazione di molti vincoli rigidi sulle dimensioni supportate e quindi la creazione di un sistema molto più adattabile a diversi usi
    - l'introduzione del supporto alle architetture multiprocessore, con conseguenze su numerosi aspetti del sistema, in particolare sui problemi di sincronizzazione

Una ulteriore evoluzione da tenere presente riguarda i terminali grafici. I sistemi Unix originariamente usavano terminali alfanumerici – i sistemi operativi gestivano solo questi. Quando i terminali grafici divennero diffusi, vennero create applicazioni ad hoc come X Windows per gestirli.; queste

funzionavano come normali processi e accedevano le interfacce Hardware grafiche e l'area di memoria RAM video. Dalla versione 2.6 Linux fornisce un'interfaccia astratta del frame buffer della scheda grafica che permette alle applicazioni di accederle senza bisogno di conoscere i dettagli fisici dell'Hardware.

Infine una nota terminologica: spesso si utilizzeranno i termini Nucleo e Kernel. In realtà questi termini sono piuttosto ambigui; talvolta si riferiscono alla parte più centrale del sistema, quella che gestisce l'esecuzione dei processi, talvolta a tutto il sistema operativo esclusi i sottosistemi più specifici di gestione della memoria, del file System e i Device Driver, talvolta a tutto il sistema operativo.

2. Il Sistema operativo come Gestore dei Processi

La funzione generale svolta da un Sistema Operativo può essere definita come la gestione efficiente dell'Hardware orientata a rendere più semplice il suo impiego da parte dei Programmi Applicativi; il modo fondamentale per raggiungere questo scopo si basa sulla nozione di processo come esecutore dei programmi applicativi. Pertanto, la funzione fondamentale del SO è la creazione e gestione dei processi.

Il supporto ai programmi applicativi si basa sulla realizzazione di **macchine virtuali**[1], più semplici da utilizzare rispetto all'Hardware, dette **processi**. Gli utilizzatori di tali macchine sono i **programmi applicativi**. Un processo è quindi una macchina virtuale che esegue un programma; per questo motivo in certi casi si fa riferimento in maniera intercambiabile al processo oppure al programma in esecuzione da parte del processo. I due concetti sono però distinti; tra l'altro, è noto che l'esecuzione di un servizio di *exec* permette di sostituire il programma in esecuzione da parte di un processo con un altro programma.
Durante la sua esecuzione il programma può richiedere al processo anche particolari funzioni dette **servizi di sistema** (**system services**). Tra i servizi fondamentali troviamo:

- i servizi di **creazione dei processi e di esecuzione dei programmi**, in particolare i servizi fork, exec, wait e exit.
- altri servizi collegati a questi riguardano la **comunicazione tra diversi processi e la segnalazione di eventi ai processi**, che non vengono trattati in questo testo.
- i **servizi di accesso ai file** (*open, close, read, write, lseek,* ecc…) tramite i quali i programmi applicativi possono non solo accedere ai file normali, ma, grazie alla nozione di **file speciale**, possono accedere anche alle periferiche. Ad esempio, la stampa su una stampante o l'interazione con il terminale sono visti come operazioni su file speciali associati alla stampante e al terminale.

Alcuni programmi applicativi sono a loro volta orientati a permettere all'utilizzatore umano un modo facile di interagire col sistema, primi fra tutti gli **interpreti di comandi** (o **Shell**) e le **interfacce grafiche** (**GUI**). E' importante tenere presente che sia le Shell che le GUI sono fondamentalmente dei particolari tipi di programmi applicativi, quindi le loro funzionalità dettagliate, pur essendo considerate spesso le funzionalità del SO, non sono altro che un modo particolare di utilizzo delle vere e proprie funzionalità del sistema, che sono definite dai servizi di sistema (figura 1).
Nel realizzare i processi il Sistema Operativo deve gestire le peculiarità dei diversi tipi di Hardware esistenti e deve adattarsi alle evoluzioni dell'Hardware che si verificano nel corso del tempo.
In base a queste premesse è logico analizzare le funzioni svolte da un SO analizzando i servizi messi a disposizione dei programmi applicativi da un lato e analizzando i meccanismi adottati dal SO per gestire la varietà dell'Hardware e la sua evoluzione dall'altro.

Processi e Thread

Nel sistema Linux i Thread sono realizzati come particolari tipi di processi, detti **Processi Leggeri** (**Lightweight Process**). Per ovviare alle ambiguità della terminologia useremo i seguenti termini:

- **Processo Leggero**: indica un processo creato per rappresentare un Thread
- **Processo Normale**: quando vogliamo indicare un processo che non è un processo leggero
- **Processo** o **Task**: per indicare un generico processo, normale o leggero (nella documentazione e nel codice Linux si usa il termine Task)

[1] le macchine create dal Software sono chiamate spesso macchine virtuali

I processi leggeri appartenenti allo stesso processo normale condividono tutti la stessa memoria, esclusa la pila. ...
Linux associa internamente ad ogni processo (normale o leggero) un diverso PID, che costituisce un identificatore univoco del processo. Dato che lo standard Posix richiede che tutti i Thread di uno stesso processo condividano lo stesso PID è necessario un adattamento tra i due modelli. Per permettere la realizzazione del modello di thread definito da Posix Linux ha introdotto la nozione di **Thread Group**. Un Thread Group è costituito dall'insieme di Processi Leggeri che appartengono allo stesso Processo Normale. In risposta alla richiesta *getpid()* tutti i processi del gruppo restituiscono il PID del *thread group leader*, cioè del primo processo del gruppo (TGID). I processi che hanno un unico thread (il thread di default) possiedono quindi un TGID uguale al loro PID.
Quindi in Linux ad ogni Processo è associata una coppia di identificatori <*PID, TGID*> dove *gettid ( )* restituisce il PID del processo, e *getpid( )* restituisce il TGID che è identico per i Processi dello stesso gruppo.

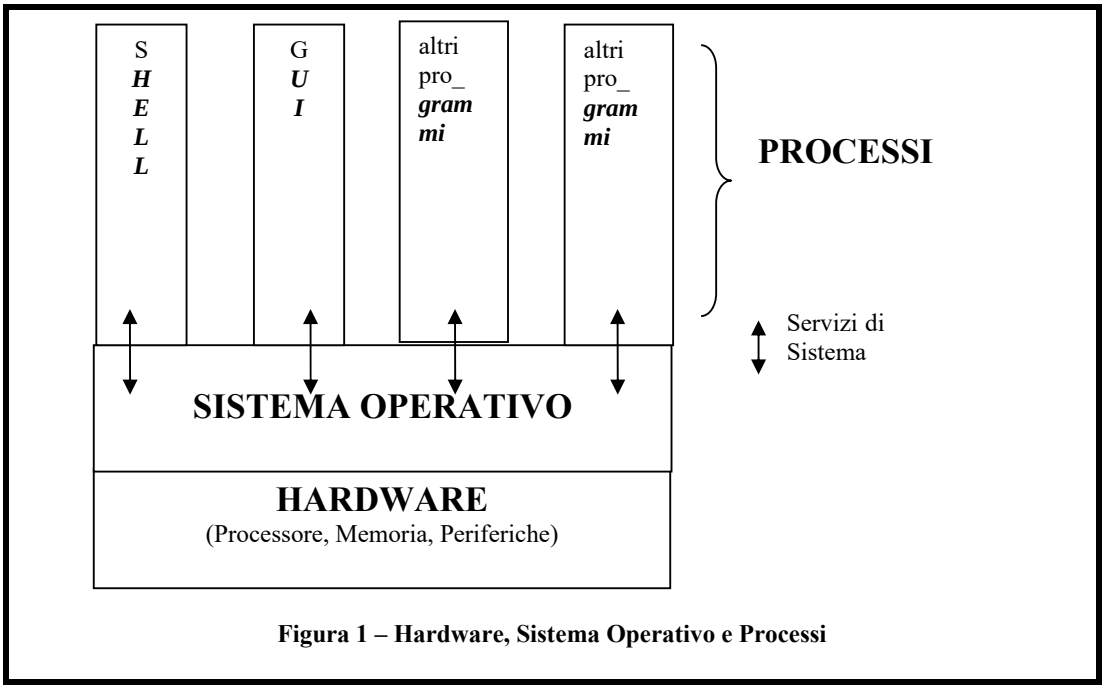

Figura 1 – Hardware, Sistema Operativo e Processi

Aspetti fondamentali della gestione dei processi
La funzione più fondamentale svolta da un sistema operativo è l'esecuzione di diversi processi in parallelo, come se esistesse un processore per ognuno di essi (virtualizzazione del processore). Per ottenere questo effetto il sistema operativo deve far eseguire al processore i programmi dei diversi processi *in alternanza*. La sostituzione di un processo in esecuzione con un altro è chiamata **Commutazione di Contesto** (**Context Switch**); con il termine **Contesto** di un processo intendiamo l'insieme di informazioni relative ad ogni processo che il SO mantiene.
La commutazione di contesto è l'operazione più importante svolta dal nucleo e richiede di risolvere due problemi fondamentali:

1. Quando deve avvenire, cioè quando è il momento di sostituire un processo in esecuzione con un altro?
2. Quale processo scegliere, tra quelli disponibili, per metterlo in esecuzione?

Si osservi che il conseguimento di un comportamento desiderato da parte dei processi dipende congiuntamente da ambedue i criteri.

L'esecuzione di un programma può essere sospesa per due motivi:

- perchè il processo è arrivato ad un punto in cui decide autonomamente di porsi in attesa, ad esempio perchè deve aspettare l'arrivo di un dato dall'esterno, oppure
- perchè il processo viene sospeso forzatamente (adesempio, a causa del completamento del tempo a sua disposizione o per soddisfare le esigenze di processi a priorità maggiore)

Il componente del SO che decide quale processo mandare in esecuzione è detto **Scheduler**. Il comportamento dello Scheduler realizza la **Politica di Scheduling** ed è orientato a garantire le seguenti condizioni:

- che i processi più importanti vengano eseguiti prima dei processi meno importanti
- che i processi di pari importanza vengano eseguiti in maniera equa; in particolare ciò significa che nessun processo dovrebbe attendere il proprio turno di esecuzione per un tempo molto superiore agli altri.

In particolare, LINUX è un sistema **time-sharing**, che fa eseguire un programma per un certo tempo, quindi ne sospende l'esecuzione (**preemption**) per permettere l'esecuzione di altri processi. In questo modo il SO tenta di garantire che il processore esegua i programmi in maniera equa, senza essere monopolizzato da un unico programma.

Sistemi multi-processore - SMP

I calcolatori moderni sono generalmente dotati di molti processori e il SO deve tener conto di questo fatto. Le architetture multi-processore evolvono continuamente e per conseguenza Linux evolve per tenerne conto. Attualmente il tipo di architettura multi-processore meglio supportata da Linux è l'architettura **SMP** (Symmetric Multiprocessing).

In un'architettura SMP esistono 2 o più processori identici collegati a una singola memoria centrale, che hanno tutti accesso a tutti i dispositivi periferici e sono controllati da un singolo sistema operativo, che li tratta tutti ugualmente, senza riservarne nessuno per scopi particolari. Nel caso di processori multi-core il concetto di SMP si applica alle singole core, trattandole come processori diversi.

L'approccio di Linux versione 2.6 al SMP consiste nell'allocare ogni task a una singola CPU in maniera relativamente statica e nello svolgere una riallocazione dei task tra le CPU solo quando, in un controllo periodico, il carico delle CPU risulta fortemente sbilanciato (*load balancing*). Un motivo di questo approccio, che potrebbe cambiare in successive versioni, è che lo spostamento di un task da una CPU a un'altra richiede di svuotare la cache (perché le cache sono generalmente strettamente collegate a una singola CPU), introducendo un ritardo nell'accesso a memoria finché i dati non sono stati caricati nella cache della nuova CPU.

Grazie a questo approccio il funzionamento di Linux può in molti aspetti essere compreso senza considerare l'esistenza di molti processori. Infatti, se consideriamo un processo eseguito da un processore, vediamo che esso non è influenzato dall'esistenza degli altri processori, se non per un numero limitato di aspetti che prenderemo in considerazione in un successivo capitolo. Ad esempio, mentre in un sistema mono-processore esiste un unico processo in esecuzione a un certo istante di tempo, detto **processo corrente**, in realtà *in Linux esiste un processo corrente per ogni processore*. Tuttavia, questo fatto è mascherato dall'esistenza di una funzione `get_current( )`, che restituisce un riferimento al descrittore del processo corrente del processore che la esegue.

Il supporto di Linux all'evoluzione delle architetture HW non si è arrestato con l'SMP; ad esempio, sono supportate le architetture NUMA (Non Uniform Memory Access), nelle quali ogni CPU accede in maniera più efficiente alcuni banchi della memoria. E' prevedibile che questa evoluzione continuerà, ma in questo testo non toccheremo più questo argomento.

Kernel non-preemptable

Linux, nella sua versione normale, applica la seguente regola: *proibire la preemption quando un processo esegue servizi del sistema operativo*. In parole diverse, questa regola viene indicata anche dicendo che il Kernel di Linux è ***non-preemptable***.

Questa regole semplifica notevolmente la realizzazione del Kernel per vari motivi che in parte verranno spiegati nel seguito.

E' però possibile compilare il nucleo con l'opzione `CONFIG_PREEMPT` per ottenerne una versione in cui il Kernel può essere preempted; lo scopo di queste versioni verrà accennato nel capitolo relativo alle politiche di scheduling e ai sistemi real-time.

Tutta la spiegazione del funzionamento del nucleo verrà fatto facendo riferimento al nucleo non-

preemptable.

Protezione del SO e dei processi
La gestione dei processi comporta anche un aspetto di limitazione delle azioni che un processo può svolgere, perché il SO deve evitare che un processo possa svolgere azioni dannose per il sistema stesso o per altri processi. Vedremo che per ottenere questo risultato è necessario il supporto di alcuni meccanismi Hardware.

3. Il sistema operativo e la gestione efficiente delle risorse Hardware

Il sistema operativo deve gestire le risorse fisiche disponibili (Hardware) in maniera efficiente.
Le risorse che il sistema operativo deve gestire sono fondamentalmente le seguenti:

- il **processore** (o i processori), che deve essere assegnato all'esecuzione dei diversi processi e del sistema operativo stesso
- la **memoria**, che deve contenere i programmi (codice e dati) eseguiti nei diversi processi e il sistema operativo stesso
- le **periferiche**, che devono essere gestite in funzione delle richieste dei diversi processi

Un aspetto di fondamentale importanza nel comprendere le caratteristiche di un sistema operativo è l'esigenza del sistema operativo di adattarsi nel tempo all'evoluzione delle tecnologie Hardware, in particolare delle periferiche. La vita di un sistema operativo infatti è lunga; in particolare, il sistema UNIX, di cui LINUX rappresenta l'ultima evoluzione, risale come già detto, al 1970. Le periferiche vengono invece riprogettate praticamente in continuazione da una moltitudine di produttori diversi, quindi è fondamentale che il sistema operativo possa essere adattato continuamente per la gestione di nuove periferiche.

Questo obiettivo pone una serie di requisiti, solo alcuni dei quali erano risolti dalla versione di Linux iniziale.
Per rendere relativamente trasparenti le caratteristiche delle periferiche rispetto ai programmi applicativi, fin dall'inizio i sistemi UNIX hanno adottato la seguente strategia:

- l'accesso alle periferiche avviene assimilandole a dei file (detti *file speciali*), in modo che un programma non debba conoscere i dettagli realizzativi della periferica stessa; ad esempio, dato che un programma scrive su una stampante richiedendo dei servizi di scrittura su un file speciale associato alla stampante, se la stampante viene sostituita con un diverso modello il programma non ne risente (ovviamente, è il diverso gestore (driver) della stampante che si occupa di gestire le caratteristiche della nuova stampante).

Per rendere semplice l'evoluzione del sistema per supportare una nuova periferica sarebbero utili due caratteristiche:

- poter aggiungere al SO il software di gestione di una nuova periferica, detto **gestore di periferica** (**device driver**) con relativa facilità;
- permettere di configurare un sistema solo con i gestori delle periferiche effettivamente utilizzate, altrimenti col passare del tempo la dimensione del SO diventerebbe enorme.

Nelle versioni iniziali di Linux questi obiettivi non erano raggiunti, perchè era necessario ricollegare l'intero sistema per includervi un nuovo driver. Successivamente, il problema è stato risolto introducendo la possibilità di inserire nel sistema nuovi moduli software, detti **kernel_modules**, senza dover ricompilare l'intero sistema.
Inoltre i kernel_modules possono essere caricati dinamicamente nel sistema durante l'esecuzione, solo quando sono necessari. Questo meccanismo nel tempo è diventato sempre più importante; in particolare questo meccanismo è il più impiegato per realizzare ed inserire nel sistema i device driver**.**
L'importanza dei moduli è attualmente tale che lo spazio di indirizzamento messo a disposizione dei moduli è doppio rispetto allo spazio di indirizzamento del sistema base.

4. Modalità di studio di Linux – notazioni utilizzate

Nei capitoli seguenti viene fornita una descrizione dei meccanismi di funzionamento di Linux. A causa delle dimensioni del sistema non è certamente possibile descrivere in poco spazio i dettagli delle funzioni e delle strutture dati dell'intero sistema. Per questo motivo verrà descritto un *modello di sistema operativo che rispecchia abbastanza fedelmente il funzionamento di Linux*, ma lo semplifica notevolmente, eliminando una serie di dettagli e di funzionalità secondarie.
In particolare lo studio segue due direttrici ortogonali:

a) la descrizione delle principali funzionalità che compongono il sistema e della loro interazione
b) la descrizione dettagliata di alcuni meccanismi specifici utilizzati da alcune funzioni

Per quanto riguarda il secondo aspetto vengono trattati in particolare i meccanismi più legati al funzionamento dell'Hardware, sia per creare un ponte con la prima parte del corso, sia perché sono quelli più difficili da comprendere per un programmatore di un linguaggio ad alto livello.
Per quanto riguarda il primo aspetto le funzionalità sono descritte se possibile utilizzando direttamente le funzioni presenti nei sorgenti di Linux, ma talvolta utilizzando delle funzioni o strutture dati che costituiscono delle astrazioni rispetto alle funzioni effettive del Software. Chiameremo quindi **funzioni** (strutture dati, costanti) **astratte** le funzioni definite in sostituzione di insiemi di funzioni (strutture dati, costanti) reali, cioè effettivamente presenti nel codice sorgente del sistema.
Le funzioni e strutture dati astratte sono facilmente riconoscibili da quelle originali, perchè sono denominate in italiano (ad esempio, lo stato di ATTESA rappresenta una costante astratta che generalizza rispetto agli stati INTERRUPTIBLE e UNINTERRUPTIBLE). Questa scelta è stata adottata in particolare quando le funzioni reali che implementano una certa funzionalità sono eccessivamente intricate rispetto all'obbiettivo primario (spesso ciò è dovuto all'evoluzione del software e alle esigenze di compatibilità) oppure sono influenzate da aspetti del sistema che no vengono trattati (come la comunicazione tra processi, ad esempio).

L'insieme delle funzioni reali di Linux e di quelle astratte è costruito in modo da mantenere la caratteristica di un sistema coerente, anche se semplificato rispetto all'originale; l'obiettivo primario dello studio è quello di capire come funziona questo sistema semplificato.

Uso dei Kernel Modules

Grazie alla vicinanza tra il sistema semplificato e quello reale è possibile supportare la comprensione del sistema semplificato eseguendo dei programmi o altri tipi di prove sul sistema reale; in particolare verranno presentati alcuni kernel modules realizzati proprio allo scopo di approfondire la comprensione del sistema. Non è invece tra gli obiettivi del testo e del corso insegnare a scrivere dei moduli ben fatti. In particolare, dato che i moduli forniti come esempi sono scritti con lo scopo di esplorare il sistema e quindi spesso volutamente *non utilizzano le funzioni di Linux che forniscono delle utili astrazioni* su alcune caratteristiche del sistema, non costituiscono esempi di come un programmatore di moduli dovrebbe operare.

Notazioni utilizzate

Nel testo sono presenti sia codice sorgente estratto direttamente dai sorgenti di Linux, sia funzioni astratte definite in sostituzione di quelle reali, sia programmi scritti come esempi. Per evitare confusione, tutti i testi che sono estratti direttamente dal codice sorgente effettivo (inclusi i nomi dei file, ecc...) sono indicati con il carattere `consolas`. Ad esempio, la struttura dati che contiene la descrizione di un processo è indicata come `struct task_struct`. I commenti aggiunti all'interno di codice sorgente originale sono anch'essi in questo carattere, ma facilmente riconoscibili da quelli originali, perchè sono in italiano.
Le funzioni astratte e i programmi scritti per esplorare il sistema e gli output prodotti sono invece in carattere Tahoma.

Scrittura di semplici Kernel Modules

Un kernel module è un programma dotato delle seguenti caratteristiche:

- può essere collegato (link) al sistema senza ricompilare l'intero sistema operativo
- diventa parte integrante del sistema, quindi può accedere alle funzioni e strutture dati del sistema
- può essere inserito e rimosso dinamicamente durante il funzionamento del sistema

L'esempio axo_hello mostra come creare un semplice kernel module che invia i seguenti messaggi, il primo quando il module viene inserito nel sistema e il secondo quando viene rimosso:

```
[ 8428.002468] Inserito modulo Axo Hello
[ 8428.012008] Rimosso modulo Axo Hello
```

I messaggi inviati dal modulo sono prefissati da un numero perchè sono messaggi del SO.
Il codice del modulo è mostrato in figura 2. I commenti lo rendono autoesplicativo.
Per mettere in funzione il modulo è necessario compilarlo, collegarlo e caricarlo nel sistema; quando non serve più si può rimuoverlo. Queste operazioni sono svolte dallo script di shell riportato in figura 3. Lo

script è costiuito da commenti che iniziano con # e da comandi interpretati dall'interprete comandi (shell).

```
/* inclusioni minimali per il funzionamento di un Kernel module */
#include <linux/init.h>
#include <linux/module.h>

/* Questa è la funzione di inizializzazione del modulo,
 *  che viene eseguita quando il modulo viene caricato */
static int __init
axo_init(void)
{
        /* printk sostiuisce printf, che è una funzione di libreria
         *  non disponibile all'interno del Kernel */
        printk("Inserito modulo Axo Hello\n");
        return 0;
}
/* Questa macro è contenuta in <linux/module.h> e comunica al
 * sistema il nome della funzione da eseguire come inizializzazione */
module_init(axo_init);

/* stesse considerazioni per axo_exit e la macro module_exit */

static void __exit
axo_exit(void)
{
        printk("Rimosso modulo Axo Hello\n");
}
module_exit(axo_exit);

/* MODULE_LICENSE informa il sistema relativamente alla licenza con la quale
 * il modulo viene rilascitao - ha effetto su quali simboli
 * (funzioni, variabili, ecc...) possono essere acceduti  */

MODULE_LICENSE("GPL");
/* Altre informazioni di identificazione del modulo */
MODULE_AUTHOR("Axo teacher");
MODULE_DESCRIPTION("prints axo hello");
MODULE_VERSION("");
```

Figura 2 – il modulo axo_hello

```
#!/bin/bash
# attiva la visualizzazione dei comandi eseguiti
        echo "build module"
# esegui compilazione e link (tramite makefile)
        make
        echo "remove and insert module"
# inserisci il modulo (ubuntu usa sudo per operazioni che richiedono diritti di amministratore)
        sudo insmod ./axo_hello.ko
# ricevi i messaggi del sistema (che normalmente non compaiono a terminale
# e inviali al file console.txt (creandolo o svuotandolo)
        dmesg|tail -1  >console.txt
# rimuovi il modulo
        sudo rmmod axo_hello
# ricevi i messaggi e appendili al file console.txt
        dmesg|tail -1  >>console.txt
```

Figura 3 - Script per la compilazione ed esecuzione del modulo

La parte più complessa di tutta l'operazione è nascosta dall'invocazione del comando "make", che esegue uno script nel quale il modulo viene compilato e collegato al sistema operativo. Questa parte è troppo complessa per essere analizzata (vedi approfondimenti alla fine del capitolo)..

5. Dipendenza dall'architettura Hardware

Linux è scritto in linguaggio C e compilato con il compilatore (e linker) ***gnu gcc***; questo fatto semplifica molto la sua portabilità sulle diverse piattaforme Hardware per le quali esiste il compilatore.
L'adattamento alle diverse architetture non è però totalmente delegabile al compilatore, perchè alcune funzioni del SO richiedono di tener conto delle peculiarità dell'Hardware sul quale sono implementate. Per questo motivo alcune funzioni del sistema sono implementate in maniera diversa per le diverse architetture e nella fase di collegamento del SO per una certa architettura viene scelta l'implementazione opportuna. Tale scelta comporta una strutturazione complessa dei file di comandi (Makefile) che guidano la compilazione. I file che contengono codice dipendente dall'architettura sono sotto la cartella di primo livello `<linux/arch>.`

Architettura x64
Nei casi in cui la spiegazione di una funzione richiederà di fare riferimento ad una particolare architettura faremo riferimento alla architettura ***x86-64, long 64-bit mode***, che chiameremo per semplicità **x64**. I file relativi all'architettura x86 si trovano nella cartella <linux/arch/x86>.

Architettura x86_64 - generalità
L'architettura ISA x86-64 è stata definita nel 2000 come evoluzione dell'architettura Intel x86 a 32 bit; il primo processore è stato prodotto da AMD nel 2003. Per ragioni storico-commerciali è nota con diversi nomi, tra i quali i più diffusi sono "AMD64", "EM64T", "x86_64" e "x86-64".
Un aspetto importante di questa architettura è la compatibilità con le numerose architetture della famiglia x86 che si sono succedute negli anni; in pratica questo significa che un processore che supporta l'ISA completa x86-64 può funzionare in ben 5 modalità diverse:

- Long 64-bit mode
- Long Compatibility mode
- Legacy Protected mode
- Legacy Virtual 8086 mode
- Legacy Real mode

Noi chiameremo x64 un processore dotato di ISA x86-64 funzionante in modalità Long 64-bit; inoltre non descriveremo completamente tale architettura, ma solamente gli aspetti rilevanti per capire il funzionamento del SO.
L'architettura x86-64 è diventata dominante nel settore dei PC personali e Server (mentre nel settore mobile è dominante l'architettura ARM). Ad esempio tra i supercomputer analizzati da TOP500 le percentuali di macchine di questa architettura nel 2013 ha superato il 90%.

6 Strutture dati per la gestione dei processi

L'informazione relativa ad ogni processo è rappresentata in strutture dati mantenute dal SO. Per il processo in esecuzione una parte del contesto, detta **Contesto Hardware,** è rappresentata dal contenuto dei registri della CPU; anche tale parte deve essere salvata nelle strutture dati quando il processo sospende l'esecuzione, in modo da poter essere ripristinata quando il processo tornerà in esecuzione.
Le strutture dati utilizzate per rappresentare il contesto di un processo sono due:

- una struttura dati, forse la più fondamentale del nucleo, che chiameremo il **Descrittore del Processo**. L'indirizzo del Descrittore costituisce un identificatore univoco del processo.
- una pila di sistema operativo (sPila) del processo. ***Si osservi che esiste una diversa sPila per ogni processo***.

Descrittore del Processo
La struttura di un descrittore è definita nel file `<linux/sched.h>` `in` figura 4 sono riportati alcuni dei campi della struttura di un descrittore (la struttura completa è molto più grande). L'ordine delle definizioni è stato modificato e alcuni commenti non sono del file originale ma sono stati aggiunti (in

italiano). Molti campi puntano a strutture dati che saranno oggetto di capitoli successivi (gestione della memoria, gestione dei files).

Questa struttura viene allocata dinamicamente nella memoria dinamica del Nucleo ogni volta che viene creato un nuovo processo. Negli esempi noi indicheremo spesso i processi con un nome che ipotizziamo sia anche il nome di una variabile contenente un riferimento al suo descrittore; ad esempio diremo: esistono 2 processi P e Q intendendo che P e Q siano 2 variabili dichiarate nel modo seguente:

- `struct task_struct * P;`
- `struct task_struct * Q;`

e quindi la notazione `P->pid` indica il campo pid del descrittore del processo P.

```
struct task_struct {
 pid_t pid;
 pid_t tgid;
 volatile long state;     /* -1 unrunnable, 0 runnable, >0 stopped */

 void *stack;  //puntatore alla sPila del task

 /* CPU-specific state of this task */
 struct thread_struct thread;

 // variabili utilizzate per Scheduling – vedi capitolo relativo
 int on_rq;
 int prio, static_prio,  normal_prio;
 unsigned int rt_priority;
 const struct sched_class *sched_class;

 // puntatori alle strutture usate nella gestione della memoria –
 //vedi capitolo relativo
 struct mm_struct *mm, *active_mm;

 // variabili per la restituzione dello stato di terminazione
 int exit_state;
 int exit_code, exit_signal;

 /* filesystem information */
       struct fs_struct *fs;
 /* open file information */
       struct files_struct *files;
…
}
```

Figura 4 – struttura del descrittore di processo

Come già detto, esiste una funzione `get_current( )` che restituisce un puntatore al task corrente della CPU che la esegue.

```
struct thread_struct {
 ...
 unsigned long sp0;
 unsigned long sp;  //puntatore alla pila di modo S (vedi prossimo capitolo)
 unsigned long usersp; // puntatore alla pila di modo U (idem)
 ...
}
```

Figura 5

Un campo di `task_struct` che riveste un significato particolare è `thread`, di tipo `struct`

`thread_struct`. Questo campo è destinato a contenere il "contesto hardware" della CPU di un processo quando non è esecuzione. Come intuibile, questa struttura è dipendente dall'Hardware, ed è definita, per l'architettura x86, nel file `Linux/arch/x86/include/asm/processor.h`. Non possiamo analizzare questa struttura senza una conoscenza dell'architettura HW, quindi ci limitiamo a vederne 3 campi che fanno riferimento al registro SP.

Per esplorare la struttura `task_struct` possiamo inserire nella routine di inizializzazione di un modulo come hello_axo una chiamata alla funzione task_explore( ), riportata in figura 6.a. La funzione mostra l'uso di `get_current()` per accedere al descrittore del processo corrente e poi visualizza il contenuto di alcuni campi.
Il risultato dell'esecuzione è mostrato in figura 6.b. L'interpretazione del risultato conferma quanto detto a proposito dei PID e TGID. L'interpretazione degli indirizzi relativi alla pila verrà svolta nel prossimo capitolo, perché dipende da aspetti dell'Hardware non ancora visti.

```
static void task_explore(void){
        struct task_struct * ts;
        struct thread_struct threadstruct;
        int pid, tgid;
        long state;
        long unsigned int vsp, vsp0, vstack, usp;
        // informazioni sui PID dei processi
        ts = get_current();
        pid = ts->pid;
        tgid = ts->tgid;
        printk("PID = %d, TGID = %d\n", pid, tgid);
        // informazioni sullo stack in task_struct
        threadstruct = ts->thread;
        vsp = threadstruct.sp;
        vsp0 = threadstruct.sp0;
        printk("thread.sp = 0x%16.16lX\n", vsp);
        printk("thread.sp0 = 0x%16.16lX\n", vsp0);
        vstack = ts->stack;
        printk( "ts-> stack = 0x%16.16lX\n", vstack);
        usp = threadstruct.usersp;
        printk( "usersp = 0x%16.16lX\n", usp);
}
```

(a) il modulo axo_task

```
[27663.251212] Inserito modulo Axo_Task
[27663.251214] PID = 7424, TGID = 7424
[27663.251215] thread.sp = 0xFFFF88005C645D68
[27663.251216] thread.sp0 = 0xFFFF88005C646000
[27663.251217] ts-> stack = 0xFFFF88005C644000
[27663.251218] usersp = 0x00007FFF6DA98C78
[27663.260995] Rimosso modulo Axo_Task
```

(b) risultato dell'esecuzione

Figura 6

Approfondimenti

Esplorazione del codice originale di Linux
I sorgenti della versione corrente di Linux sono disponibili su GitHub all'indirizzo https://github.com/torvalds/linux. La struttura delle cartelle è gerarchica. Nei capitoli seguenti quando viene introdotta una funzione generalmente viene indicato il file nel quale è definita, talvolta con tutto il

suo pathname che permette di individuarla. Ad esempio, la struttura `struct task_struct` si trova in `linux/kernel/sched/sched.h` .
Spesso per analizzare il codice sorgente conviene utilizzare dei servizi di cross-reference, che permettono di navigare il codice passando da una definizione all'altra. Ad esempio, se cerchiamo con un motore di ricerca "linux task_struct" troveremo tra gli altri una indicazione tipo "**Linux/include/linux/sched.h - Linux Cross Reference - Free Electrons**" che ci porta a una pagina nella quale è visualizzato il codice del file `sched.h` che la contiene; cliccando sulle diverse definizioni si accede a diversi files che le contengono.
La lettura del codice originale non è facile, non solo per la complessità del sistema, ma anche per l'uso sofisticato del linguaggio C (con alcune estensioni del compilatore gnu rispetto allo standard) e del preprocessore. Inoltre, in alcuni punti sono presenti pezzi di codice in Assembler; l'inserimento di Assembler all'interno di codice C segue le regole del "GNU Inline ASM".

Compilazione condizionata
Nel codice di molte funzioni legate all'architettura x86 esistono porzioni la cui compilazione è condizionata dal modo di funzionamento. La compilazione condizionata di queste parti in generale è individuata nel modo indicato sotto.
Il preprocessore ovviamente opera in base al valore che è stato assegnato alle costanti `CONFIG_X86_32` e `CONFIG_X86_64` al momento della compilazione.

```
#ifdef CONFIG_X86_32
 //codice a 32 bit – non ci interessa
#else
 //codice a 64 bit – ci interessa
#endif
#ifdef CONFIG_X86_64
 //codice a 64 bit – ci interessa
#endif
```

Makefile dei moduli
Il makefile utilizzato per compilare il modulo axo_hello è riportato sotto. Per comprenderlo a fondo è necessario conoscere il meccanismo dei makefile e di compilazione del sistema operativo. Anche senza comprenderlo completamente, il makefile è usabile per compilare dei moduli

```
# modulo da compilare; viene cercato automaticamente
# un sorgente con lo stesso nome e .c come estensione
obj-m := axo_hello.o

# KDIR indica la directory dei sorgenti del Kernel -
# $(shell uname -r) determina la versione corretta del sistema
KDIR  := /lib/modules/$(shell uname -r)/build

# PWD indica la directory contenente i moduli da linkare
PWD   := $(shell pwd)

# il comando effettivo - troppo complesso per essere spiegato
# fa riferimento alle convenzioni di compilazione del Kernel
# linka i moduli presenti in PWD con i simboli di KDIR  */
default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
```

Makefile del modulo axo_hello