POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Il Nucleo del Sistema Operativo

N1 – Introduzione a Linux

LINUX

Kernel di sistema operativo libero, basato su Unix, sviluppato nel 1991 da Linus Torvalds (Helsinki)

- compatibile con specifiche Unix
- eseguibile su qualsiasi PC e piattaforme hardware
- evolve con contributo di sviluppatori da tutto il mondo

E' disponibile in circa 300 diverse distribuzioni (applicazioni diverse, veste grafica diversa, nomi diversi, medesimo kernel Linux)

Linux è soltanto il nome del **kernel** e non del sistema operativo e rappresenta infatti solo una parte di esso.

Inteso come sistema operativo, è da considerarsi il completamento del sistema GNU con il kernel sviluppato da Linus Torvalds.

Perciò il termine più appropriato per riferirsi al sistema operativo è **GNU/Linux**.

Applicazioni di Linux

PC: campo dominato dal sistema Windows, Linux ha avuto un successo parziale

Server: settore di maggior successo di Linux, che costituisce il sistema dominante nella gestione dei Server

Sistemi embedded (special purpose): l'impiego di Linux è in continua crescita

Sistemi Real-Time (deadline): in origine Linux non era in grado di supportare i sistemi real-time, ma il suo adattamento è molto progredito e in continua evoluzione

Sistemi Mobile: Linux è alla base di Android, uno dei più diffusi sistemi operativi mobile

Evoluzione

anni 70 - Sistemi UNIX

1991 – Linux viene sviluppato per portare Unix su Personal Computer in architettura Intel 386

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 (Completely Fair Scheduler)
3.10	2013	15.803	
4.0	2015		

Evoluzione (2)

Aumento quasi esponenziale della quantità di codice di Linux dovuto in primo luogo dalla crescita del numero di dispositivi periferici nuovi supportati (gestori di periferiche 50% del codice complessivo)

Tra la versione 2.0 e la 2.6 aumento della complessità del sistema dovuto a due fenomeni:

- sostituzione delle strutture statiche (array) delle prime versioni di Linux con *strutture dinamiche*, con conseguente eliminazione di molti vincoli rigidi sulle dimensioni supportate (sistema più adattabile a diversi usi)
- introduzione del supporto alle *architetture multiprocessore*, con conseguente aumento della complessità relativo alla gestione di numerosi aspetti del sistema, in particolare sui problemi di sincronizzazione

Evoluzione (3)

I sistemi Unix originariamente usavano *terminali alfanumerici*

Diffusione dei ***terminali grafici***

creazione di applicazioni ad hoc come X Windows per gestirli
che funzionavano come normali processi e accedevano le interfacce Hardware grafiche e l'area RAM video

Dalla versione 2.6 Linux fornisce un'***interfaccia astratta del frame buffer*** della scheda grafica che permette alle applicazioni di accedere senza bisogno di conoscere i dettagli fisici dell'Hardware

→ possibilità di applicazioni grafiche più complesse

Funzione principale di Linux

Realizzazione di un ambiente di esecuzione dei programmi applicativi costituito da un insieme di processi

- ogni processo esegue un programma sequenzialmente
- diversi processi procedono «in parallelo» (creazione di processi e esecuzione di programmi)
- un processo non può disturbare l'esecuzione di un altro processo – isolamento dei processi
- le ***risorse condivise*** tra processi (ad esempio le periferiche) sono gestite in maniera controllata centralmente

  →un processo non può accedere direttamente a tali risorse, ma deve richiedere i servizi del SO (system services)

  →i servizi devono fornire un modello semplice delle risorse gestite (es. file speciali per rappresentare le periferiche)

Funzioni di Linux

realizzazione del parallelismo tra processi

Linux è un S.O. multiprogrammato ***time-sharing*** (divisione di tempo)

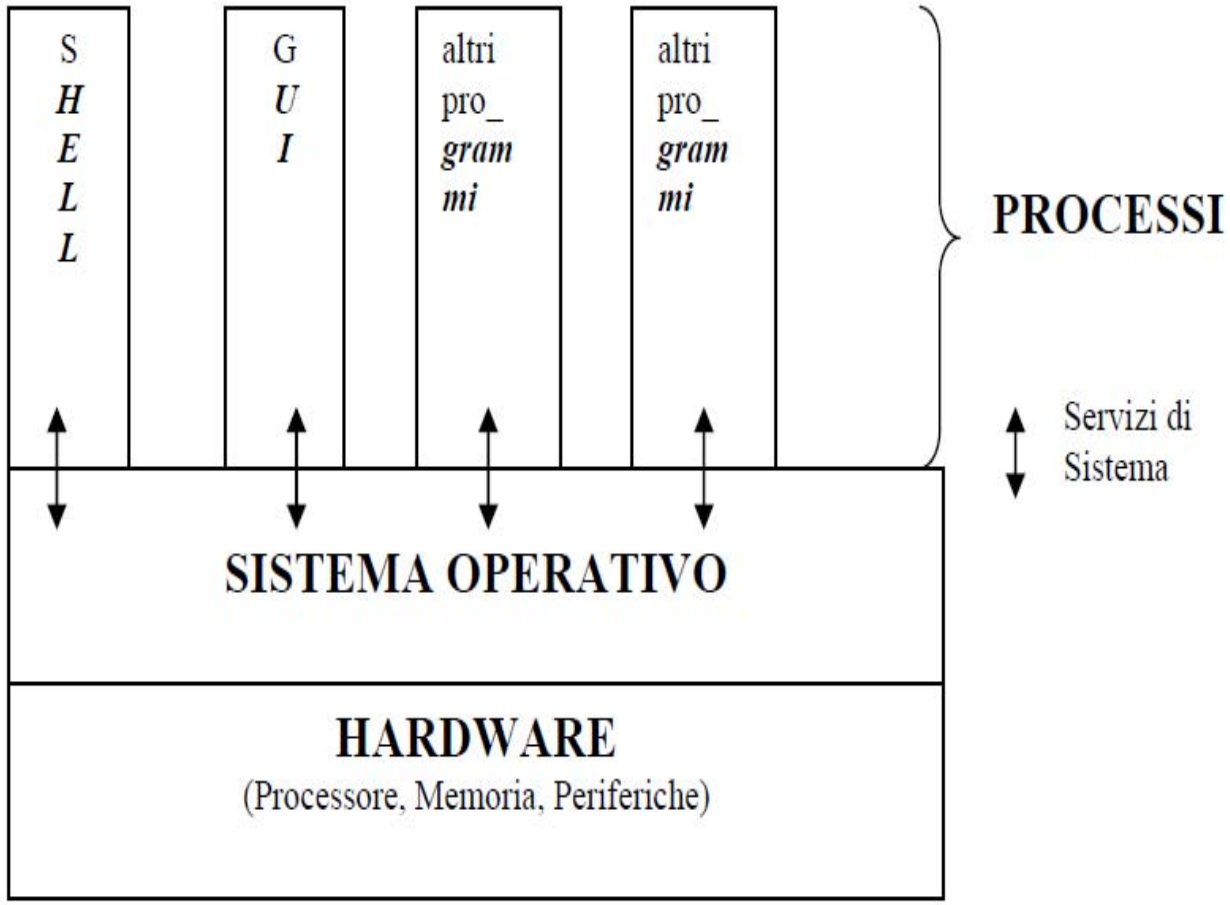
la virtualizzazione del parallelismo è ottenuta facendo eseguire *in alternanza* i diversi processi dall'unico processore

- a ogni processo è assegnato un ***quanto di tempo***
- alla scadenza del quanto di tempo il processo viene sospeso dall'esecuzione (***preemption***) e un nuovo processo può utilizzare il processore

Un processo può essere sospeso dall'esecuzione:

- allo ***scadere del quanto tempo*** o anche per soddisfare le esigenze di processi a priorità maggiore
- per ***sospensione volontaria*** dopo aver richiesto un servizio di sistema (ad esempio per eseguire un'operazione di ingresso / uscita)

Processi e Sistema Operativo



Processi e Thread

Nel sistema Linux i Thread sono realizzati come particolari tipi di processi, detti ***Processi Leggeri*** (**Lightweight Process**)

Terminologia:

- **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)

I processi leggeri appartenenti allo stesso processo normale condividono tutti la stessa memoria, esclusa la pila

Processi e Thread (2)

***Thread* (*Processo leggero*):** esecutore virtuale completo caratterizzato da

Risorse proprie
- PC
- Registri di CPU
- SP e Stack
- Stdin/Stdout/Stderr

Risorse condivise con gli altri thread dello stesso processo
- Memoria (codice, dati e segmento di sistema)

***Processo normale*:** esecutore virtuale completo caratterizzato da
- PC
- Registri di CPU
- Stack
- Stdin/Stdout/Stderr
- Memoria (codice, dati e segmento di sistema)

PID e TGID

Linux associa internamente ad ogni processo (normale o leggero) un diverso PID

Lo standard Posix richiede che tutti i Thread di uno stesso processo condividano lo stesso PID → è necessario un adattamento tra i due modelli:

- un **Thread Group** è costituito dall'insieme di Processi Leggeri che appartengono allo stesso Processo Normale
- ogni processo possiede un **TGID** (*Thread Group ID*) oltre al **PID**
- il TGID di un processo è uguale al PID del *thread group leader*, cioè del primo processo del gruppo (quello col thread di default)
- i processi che hanno un unico thread (il thread di default) hanno quindi un TGID uguale al loro PID.

Ad ogni Processo è associata la coppia di identificatori ***<PID, TGID >*** dove

- *gettid ( )* restituisce il PID del processo
- *getpid( )* restituisce il TGID che è identico per i Processi dello stesso gruppo

Gestione dei processi

La sostituzione di un processo in esecuzione con un altro è chiamata **Commutazione di Contesto** (***Context Switch***)

con il termine **Contesto** di un processo si intende l'insieme di informazioni relative ad ogni processo che il SO gestisce

- quando un processo *è in esecuzione* una parte del suo contesto è nei registri di CPU (PC e altri registri) e una parte è in memoria
- quando un processo *non è in esecuzione* tutto il suo contesto è in memoria, cioè è stato «salvato» dal SO per poter essere ripristinato quando il processo tornerà in esecuzione

Una commutazione di contesto prevede quindi

- un processo che lascia l'esecuzione, e il suo contesto deve essere in parte salvato
- un processo che entra in esecuzione e il suo contesto deve essere in parte ripristinato

Lo Scheduler

Il componente del SO che decide quale processo mandare in esecuzione è detto **Scheduler**

Obbiettivi della **Politica di Scheduling**

- 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 (bilanciamento tra processi CPU_bound e I/O_bound: ad es. *preemption*)

Lo scheduler è anche il componente responsabile di attuare la commutazione di contesto

Sistemi multiprocessore - SMP

Il tipo di architettura multi-processore attualmente meglio supportata da Linux è l'architettura **SMP** (Symmetric Multiprocessing)

- 2 o più processori identici collegati a una singola memoria centrale,
- hanno tutti accesso a tutti i dispositivi periferici
- sono controllati da un singolo sistema operativo e vengono considerati identici (cioè nessun processore è riservatoper scopi particolari)
- nel caso di processori multi-core il concetto di SMP si applica ai singoli core, considerandoli come processori diversi

L'approccio di Linux al SMP consiste nell'allocare ogni task a una singola CPU:

- allocazione statica
- riallocazione dei task tra le CPU per *bilanciamento del carico di lavoro*, cioè se in un controllo periodico, il carico delle CPU risulta fortemente sbilanciato

LINUX e SMP

Gestione statica dei task

- ✓ lo spostamento di un task da una CPU a un'altra richiede di svuotare la cache, che sono generalmente strettamente collegate a una singola CPU
- ✓ lo spostamento introduce un ritardo nell'accesso a memoria finchè i dati non sono stati caricati nella cache della nuova CPU

Linux può essere compreso (per molti aspetti) senza considerare l'esistenza di più processori

- un processo è eseguito da un solo processore,
- non è influenzato dall'esistenza degli altri processori,
- parleremo di **processo corrente** (processo correntemente in esecuzione), anche se in realtà *in Linux esiste un processo corrente per ogni processore*
- esiste una funzione `get_current( )` che restituisce il riferimento al processo corrente del processore che la esegue

Kernel non-preemptable

il Kernel di Linux è ***non-preemptable***, cioè:

è proibita la preemption quando un processo esegue codice del sistema operativo

Questa regola semplifica notevolmente la realizzazione del Kernel per vari motivi

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 è orientato ai sistemi real-time.

noi faremo riferimento solo alla versione non-preemptable

Protezione dei processi

il SO deve evitare che un processo possa svolgere azioni dannose per il sistema stesso o per altri processi

La gestione dei processi comporta quindi anche un aspetto di limitazione delle azioni che un processo può svolgere,

per ottenere questo risultato è necessario il supporto di alcuni meccanismi Hardware.

Gestione efficiente delle risorse HW

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

il sistema operativo deve adattarsi nel tempo all'evoluzione delle tecnologie Hardware, in particolare delle periferiche

Adattamento di Linux all'evoluzione delle periferiche (1)

Per rendere relativamente trasparenti le caratteristiche delle periferiche rispetto ai programmi applicativi:

- l'accesso alle periferiche avviene assimilandole a dei file
- un programma non ha necessità di 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
- è il diverso **gestore** (**driver**) della stampante che si occupa di gestire le caratteristiche della nuova stampante

Adattamento di Linux all'evoluzione delle periferiche (2)

Per rendere semplice l'evoluzione del sistema per supportare una nuova periferica sono utili 2 caratteristiche

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

Linux fornisce la possibilità di inserire nel sistema nuovi moduli software, detti ***kernel_modules***, senza dover ricompilare l'intero sistema

•i kernel_modules possono essere caricati dinamicamente nel sistema durante l'esecuzione, solo quando sono necessari

•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

Kernel modules – esempio axo_hello

```
#include <linux/init.h>
#include <linux/module.h>

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

/* idem per exit */
static void __exit
axo_exit(void)
{   printk("Rimosso modulo Axo Hello\n");   }
module_exit(axo_exit);
```

Compilazione ed esecuzione del modulo

```
# esegui compilazione e link (tramite makefile)
# il makefile è complesso, perché deve linkare le
# funzioni del SO
# produce axo_hello.ko (ko = kernel module) – è
# l'equivalente di un eseguibile
# per il sistema operativo
make
# inserisci il modulo, cioè caricalo nel sistema
#(ubuntu usa sudo come interprete comandi
# per operazioni che richiedono diritti di amministratore)
sudo insmod ./axo_hello.ko
# rimuovi il modulo quando non serve più
sudo rmmod axo_hello
```

Linux semplificato

Consideriamo un ***modello di sistema operativo*** *che rispecchia abbastanza fedelmente il funzionamento di Linux*, ma lo semplifica notevolmente sia relativamente alle funzioni che alle strutture dati, eliminando una serie di dettagli e di funzionalità secondarie …

- **funzioni astratte** le funzioni (strutture dati, costanti) definite in sostituzione di insiemi di funzioni reali, cioè effettivamente presenti nel codice sorgente del sistema
- riconoscibili perché denominate in italiano
  - ad esempio: stato di ATTESA sostituisce INTERRUPTIBLE e UNINTERRUPTIBLE

Dipendenza dall'Architettura HW

Linux è scritto in linguaggio C e compilato con il compilatore (e linker) ***gnu gcc***

→ portabilità sulle diverse piattaforme HW per le quali esiste il compilatore

Alcune *funzioni dipendono comunque dalle peculiarità dell'HW* e sono implementate in maniera diversa per le diverse architetture (possono includere codice assembler)

Nella fase di link del SO per una certa architettura viene scelta l'implementazione opportuna

i file che contengono codice dipendente dall'architettura sono sotto la cartella di primo livello <**linux/arch**>

⇨ strutturazione complessa dei file di comandi (Makefile) che guidano la compilazione

Architettura x64

noi faremo riferimento, ove necessario, all'architettura ***x86-64, long 64-bit mode***

- architettura ISA definita nel 2000 - evoluzione dell'architettura Intel x86 a 32 bit
- compatibile con le numerose architetture x86 che si sono succedute negli anni

sorgenti LINUX relativi all'architettura x86 si trovano in <linux/arch/x86>

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

chiameremo **x64** un processore dotato di ISA x86-64 funzionante in modalità Long 64-bit

è l'architettura assolutamente dominante al momento nei PC e nei server

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/salvare il contesto di un processo sono

–una struttura dati, forse la più fondamentale del nucleo, chiamata ***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 - file <linux/sched.h>

Questa struttura viene allocata dinamicamente nella memoria dinamica del Nucleo ogni volta che viene creato un nuovo processo

Negli esempi indicheremo spesso i processi con un nome

- ipotizziamo che sia il nome di una variabile contenente un riferimento al suo descrittore
- ad esempio diremo: esistono due processi P e Q intendendo che P e Q siano due 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

struttura del descrittore di processo

volatile: qualificatore che indica al compilatore che il valore della variabile può cambiare in modo asincrono (e.g. multithreading, memory-mapped I/O) rispetto alle «azioni» previste nel modulo in compilazione .

```
struct task_struct {
    pid_t pid;      pid_t tgid;
    volatile long state;   // -1 unrunnable, 0 runnable, >0 stopped
    void *stack;           //puntatore alla dim max sPila del task

    struct thread_struct thread;  //strutt. che contiene il contesto HW
      ……
     // variabili utilizzate per Scheduling –…

    struct mm_struct *mm
     //puntatori alle strutture usate nella gestione della memoria

     // variabili per la restituzione dello stato di terminazione _ …

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

la struttura per il contesto HW (solo puntatori alla pila)

struttura dipendente dall'HW con definizione diversa per ogni architettura

```
struct thread_struct {
 ...
//puntatore alla base della pila di sistema operativo sPila del processo
 unsigned long sp0;
//puntatore alla posizione corrente della pila di sistema operativo del processo
unsigned long sp;
unsigned long usersp; // puntatore alla pila di modo U (idem)
 ...
}
```

Accesso al descrittore tramite un kernel module

Inseriamo in axo_hello una chiamata alla seguente funzione

```
static void task_explore(void){
   struct task_struct * ts;
   struct thread_struct threadstruct;
   int pid, tgid;
   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
   threadstruct = ts->thread;
   vsp = threadstruct.sp; vsp0 = threadstruct.sp0;
    printk(“…”, vsp); printk(“…", vsp0);

   vstack = ts->stack;  printk( “…", vstack);
   usp = threadstruct.usersp; printk( “…", usp);
}
```

Risultato di esecuzione del modulo

la stampa del risultato è fatta dal SO eseguendo il modulo: per questo i messaggi hanno la numerazione dei messaggi di sistema (log di sistema)

la stampa del PID e del TGID può ovviamente essere ottenuta anche con un programma applicativo che usa i servizi opportuni (get_pid, ecc…)

gli indirizzi della pila di sistema invece non potrebbero essere ottenuti da un normale programma; la loro interpretazione verrà discussa più avanti

```
[27663.251212] Inserito modulo Axo_Task
[27663.251214] PID = 7424, TGID = 7424
[27663.251215] thread.sp = 0xFFFF 8800 5C64 5D68
[27663.251216] thread.sp0 = 0xFFFF 8800 5C64 6000
[27663.251217] ts-> stack = 0xFFFF 8800 5C64 4000
[27663.251218] usersp = 0x0000 7FFF 6DA9 8C78
[27663.260995] Rimosso modulo Axo_Task
```