Il Sistema Operativo LINUX
Aspetti Generali

Applicazioni del Sistema Operativo (*SO*) Linux

- ***PC***: in questo campo, dominato dal sistema operativo Windows, il sistema operativo Linux ha avuto un successo parziale
- **server**: questo è il settore di maggior successo di Linux, che costituisce il *SO* dominante nella gestione dei server
- **sistemi embedded**: in questo settore l'impiego di Linux è in continua crescita
- **sistemi *REAL-TIME***: il *SO* Linux in origine non era in grado di supportare i sistemi real-time, ma il suo adattamento per supportarli è molto progredito e in continua evoluzione
- **sistemi mobili**: il *SO* Linux è alla base del sistema operativo Android, uno dei *SO* più diffusi per dispositivi mobili

Evoluzione di Linux - I

- anni ‘70 – sviluppo dei Sistemi Operativi di famiglia *UNIX*
  - Free *BSD* (Berkeley), System V, *HP* Unix, *SUN* Solaris, *DEC* Ultrix, e altri ancora
- 1991 – Linux viene sviluppato come modello di *SO* tipo Unix per *PC* Intel 386

VERSIONE	ANNO	RIGHE DI CODICE C (in K)	NOTE
0.01	1991	10	prima versione di Linux
0.95	1992		X-Windows system (*GUI* a finestre)
1.0	1994	176	
2.0	1996		
2.2.0	1999	1.800	
2.2.13	1999		inizio dell’uso come «server enterprise»
2.4.0	2001	3.377	
2.6.0	2003	5.929	
2.6.15	2006		introduzione dello scheduler *CFS*
3.10	2013	15.803	
4.0	2015		

Evoluzione di Linux - II

- 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 dello 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

Evoluzione di Linux - III

- un'ulteriore evoluzione da tenere presente riguarda i terminali grafici
- originariamente i sistemi Unix usavano terminali alfanumerici – i sistemi operativi gestivano solo questi
- quando i terminali grafici si diffusero, vennero create applicazioni *ad hoc* come *X* Windows per gestirli
- queste applicazioni funzionavano come normali processi e accedevano alle interfacce Hardware grafiche e all'area *RAM* video
- a partire dalla versione 2.6, il sistema Linux fornisce un'interfaccia astratta del «frame buffer» (memoria) della scheda grafica, che permette alle applicazioni di accedere a tale buffer senza bisogno di conoscere i dettagli fisici dello Hardware
  - possibilità di applicazioni grafiche più complesse

Funzione Principale di Linux

Realizzare l'ambiente di esecuzione dei programmi applicativi costituito da un insieme di processi in modo da garantire che:

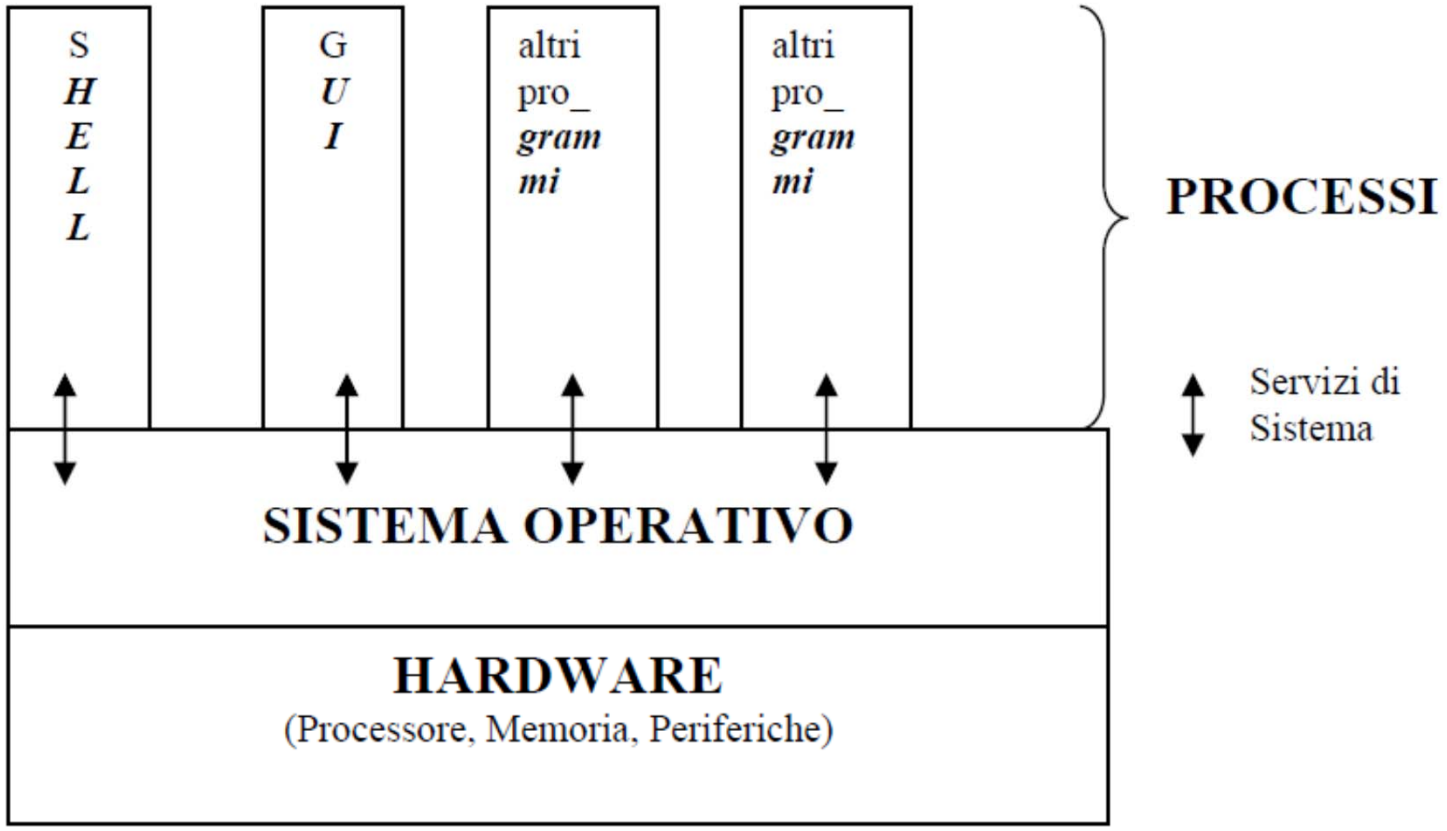
- ogni processo esegua sequenzialmente un programma
- diversi processi possano procedere «in parallelo»
- un processo non disturbi mai l'esecuzione di un altro processo – isolamento dei processi
- le risorse condivise tra processi siano gestite con controllo centrale (p. es. le periferiche)
  - un processo non può accedere direttamente a tali risorse, ma deve richiedere i servizi del *SO* (*system services*)
  - qualunque accesso alle risorse da parte del processo, senza passare per il *SO*, sia impossibile
  - i servizi devono fornire un modello semplice delle risorse gestite, come per esempio i file speciali utilizzati per rappresentare le periferiche
- la gestione delle periferiche sia efficiente, per esempio che le periferiche siano mantenute attive quando è possibile

Funzioni di Linux

Realizzare parallelismo tra processi, secondo il modello seguente:

- il sistema Linux è un *SO* multi-programmato con divisione di tempo o ***time-sharing***
- la virtualizzazione del parallelismo è ottenuta facendo eseguire in alternanza i diversi programmi dall'unico processore (a turno in modo più o meno uniforme ed equo)
- a ogni programma è assegnato un **quanto di tempo** (di solito qualche decina di ms)
- alla scadenza del quanto di tempo il programma viene sospeso dall'esecuzione (*preemption*) e un nuovo programma può passare a utilizzare il processore
- il processo può essere sospeso dall'esecuzione per i due motivi seguenti:
  - allo scadere del quanto di tempo
  - per sospensione volontaria dopo avere richiesto un servizio di sistema, per esempio per eseguire un'operazione di ingresso / uscita

Processi e Sistema Operativo



Processo e Thread

- nel sistema operativo Linux, i thread sono realizzati come un particolare tipo di processo, detto **processo leggero** (***lightweight process***)
- terminologia per i processi:
  - **processo leggero**: indica un processo creato per rappresentare un thread
  - **processo normale**: quando si vuole indicare un processo che non è un processo leggero
  - **processo** o **task**: per indicare un processo generico , normale o leggero
  - nella documentazione e nel codice di Linux si usa il termine **task**
- tutti i processi leggeri (ossia i thread) appartenenti allo stesso processo normale condividono la stessa memoria, esclusa la pila

PID e TGID

- internamente Linux associa a ciascun processo (normale o leggero) un *PID* diverso
- tuttavia lo standard *POSIX* richiede che tutti i thread di uno stesso processo condividano lo stesso *PID*, dunque è necessario adattare i due modelli:
  - un **thread group** è costituito dall'insieme di processi leggeri che appartengono allo stesso processo normale
  - ciascun processo possiede un *TGID*, oltre al *PID*
  - il *TGID* di un processo è uguale al *PID* del *thread group leader*, cioè del primo processo del gruppo (quello con il thread di default – *main*)
  - pertanto i processi che hanno un solo thread (il thread di default – *main*) possiedono un *TGID* uguale al loro *PID*
- a ciascun processo è associata una 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 a ciascun processo che il *SO* gestisce
- l'esecuzione di un programma può essere sospesa per due motivi:
  - perché il processo è arrivato a un punto dove decide autonomamente di porsi in attesa, per esempio per aspettare l'arrivo di un dato dall'esterno
  - perché il processo viene sospeso forzatamente, per esempio a causa del completamento del tempo a sua disposizione o per soddisfare le esigenze di processi a priorità maggiore

Lo Scheduler

- il componente del *SO* che decide quale processo mandare in esecuzione è chiamato **scheduler**
- obiettivi 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
- lo scheduler è anche il componente cha ha la responsabilità di attuare la commutazione di contesto

Sistemi Multiprocessore - *SMP*

- le architetture multi-processore evolvono continuamente, e di conseguenza anche il sistema operativo Linux evolve per tenerne conto
- ad oggi l'architettura multi-processore meglio supportata da Linux è il modello ***SMP*** (*Symmetric MultiProcessing*), con queste caratteristiche generali:
  - due o più processori identici sono connessi a una sola memoria centrale
  - tutti i processori hanno accesso a tutti i dispositivi periferici
  - tutti i processori sono controllati da un singolo sistema operativo, che li tratta ugualmente, senza riservarne nessuno per scopi particolari
  - per i processori *multi-core* (cioè con due o più *CPU* su un solo chip eventualmente con cache condivisa), il concetto di *SMP* si applica ai singoli core, trattandoli come processori diversi
- l'approccio di Linux a *SMP* consiste nell'allocare ciascun task a una singola *CPU*:
  - allocazione statica
  - riallocazione dei task tra le *CPU* solo quando, in un controllo periodico, il carico delle *CPU* risultasse fortemente sbilanciato (*load balancing*)

Linux e *SMP*

- un motivo dell'approccio relativamente statico di Linux:
  - lo spostamento di un task da una *CPU* a un'altra richiede di svuotare la cache, dato che generalmente le memorie cache sono strettamente connesse a una singola *CPU*
  - così introducendo un ritardo nell'accesso a memoria fino a quando i dati sono stati caricati nella cache della nuova *CPU*
- per molti aspetti, il *SO* Linux può essere compreso senza considerare l'esistenza di molti processori, per i motivi seguenti:
  - un processo è eseguito da un solo processore
  - il singolo processore non è influenzato dall'esistenza degli altri processori
  - pertanto si parlerà di **processo corrente**, anche se in realtà in Linux *esiste un processo corrente per ciascun processore*
  - esiste una funzione *get_current ( )* che restituisce il processo corrente del processore che esegue tale funzione

Nucelo (*Kernel*) non-preemptable

- il nucleo (*kernel*) di Linux è di tipo ***non-preemptable*** (non interrompibile), cioè:
  la preemption è vietata quando un processo esegue codice del SO
- questa regola semplifica notevolmente la realizzazione del nucleo, per vari motivi
- però è possibile compilare il nucleo del *SO* con l'opzione CONFIG_PREEMPT per ottenerne una versione dove il nucleo può essere «preempted»
- queste versioni sono orientate ai sistemi real-time
- qui si farà riferimento solo alla versione non-preemptable di Linux

Protezione dei Processi

- il *SO* deve evitare che un processo possa svolgere azioni dannose nei confronti del sistema stesso o per altri processi
- pertanto la gestione dei processi comporta anche un aspetto di limitazione delle azioni che un processo può svolgere
- per ottenere questo risultato è necessario il supporto di alcuni meccanismi hardware

SO e Gestione dello *HW*

- fondamentalmente le risorse che il *SO* deve gestire sono le seguenti:
  - il **processore** (o i processori), che va assegnato (in generale a turno) all'esecuzione dei diversi processi e del sistema operativo stesso
  - la **memoria**, che deve contenere sia i programmi (cioè codice e dati) eseguiti nei diversi processi, sia il sistema operativo stesso
  - le **periferiche**, che vanno gestite in funzione delle richieste dei diversi processi
- il sistema operativo deve adattarsi nel tempo all'evoluzione delle tecnologie hardware, in particolare a quelle delle periferiche

Adattamento di Linux all'Evoluzione delle Periferiche – I

- per rendere relativamente trasparenti le caratteristiche delle periferiche rispetto ai programmi applicativi, occorre quanto segue:
  - l'accesso alle periferiche avviene assimilandole a dei file
  - un programma non ha necessità di conoscere i dettagli realizzativi della periferica stessa
  - per esempio, dato che un programma scrive su una stampante richiedendo dei servizi di scrittura su un file speciale associato alla stampante stessa, se la stampante viene sostituita da una di modello diverso, il programma non ne risente
  - infatti è il diverso **gestore** (**driver**) della stampante che si occupa di gestire le caratteristiche della nuova stampante

Adattamento di Linux all'Evoluzione delle Periferiche – II

- per semplificare e agevolare l'evoluzione del sistema operativo, onde permettergli di supportare una nuova periferica, sono utili due caratteristiche:
  - potere aggiungere con relativa facilità al *SO* il software di gestione di una nuova periferica, chiamato **gestore di 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
- il *SO* Linux fornisce la possibilità di inserire nel sistema nuovi moduli software, chiamati ***kernel_module***, senza necessità di ricompilare l'intero sistema
- i kernel_module possono essere caricati dinamicamente nel sistema durante l'esecuzione, solo quando sono necessari
- attualmente l'importanza dei kernel_module è tale che lo spazio di indirizzamento di memoria messo a loro disposizione è doppio rispetto allo spazio di indirizzamento del sistema base (cioè fondamentalmente del nucleo)

Kernel Module – Esempio “axo_hello”

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

/* funzione di inizializzazione del modulo, eseguita quando viene caricato  */
static int __init
axo_init (void) {
   /* printk sostiuisce printf, non disponibile all'interno del kernel  */
   printk ("Inserito modulo Axo Hello.\n");
   return 0;
} /* axo_init */

/* comunica al sistema il nome della funzione da eseguire per inizializzare */
module_init (axo_init);

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

module_exit (axo_exit);
```

Compilazione ed Esecuzione del Modulo “axo_hello”

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

Un Modello di Sistema Operativo: Linux semplificato

- è certamente impossibile descrivere in poco spazio e tempo tutti i dettagli delle funzioni e delle strutture dati dell'intero sistema operativo Linux
- pertanto qui si considera 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:
  - talvolta si utilizzano funzioni o strutture dati che costituiscono delle astrazioni rispetto alle funzioni effettive del software
  - si chiameranno **funzioni astratte** certe funzioni (e anche le rispettive strutture dati e costanti) definite in sostituzione di funzioni o insiemi di funzioni reali, cioè effettivamente presenti nel codice sorgente del sistema
  - le funzioni astratte saranno riconoscibili perché sono denominate in italiano
- per esempio, stato di ATTESA (costante astratta) sostituisce INTERRUPTIBLE e UNINTERRUPTIBLE (costati reali)

Dipendenza all'Architettura *HW*

- il sistema operativo Linux è scritto in linguaggio C ed è compilato tramite il compilatore (e linker) ***GNU gcc***
- ciò assicura la portabilità del sistema operativo sulle diverse piattaforme hardware dove esiste il compilatore
- però alcune funzioni dipendono dalle peculiarità dello hardware e sono implementate in modo differente per le diverse architetture
- spesso queste funzioni includono codice macchina simbolico (*inline assembler*)
- nella fase di collegamento (*linking*) del *SO* per una certa architettura, viene scelta l'implementazione opportuna
- i file di comando (*makefile*) che guidano la compilazione sono strutturati in modo piuttosto complesso
- in Linux i file che contengono codice dipendente dall'architettura si trovano nella cartella di primo livello <**linux/arch**>, e nelle sotto-cartelle ivi contenute

Architettura *x64*

- qui si farà riferimento, dove necessario, all'architettura ***x86-64, long 64-bit mode***
- i sorgenti C di *Linux* relativi all'architettura *x86* si trovano in <linux/arch/x86>
- architettura *ISA* definita nel 2000, come evoluzione dell'architettura Intel *x86* a 32 bit
- compatibile con le numerose architetture *x86* che si sono succedute negli anni
- l'architettura *x86-64* può funzionare in ben cinque modalità diverse:
  - Long 64-bit mode
  - Long Compatibility mode
  - Legacy Protected mode
  - Legacy Virtual 8086 mode
  - Legacy Real mode
- qui si chiamerà ***x64*** un processore dotato di *ISA x86-64* funzionante in modo *Long 64-bit*
- al momento questa è l'architettura decisamente dominante nei sistemi PC e nei server

Strutture Dati per la Gestione dei Processi

- l'informazione relativa a ciascun processo è rappresentata in un complesso di strutture dati mantenute dal *SO*
- per il processo in esecuzione una parte del contesto, chiamata **contesto hardware,** è rappresentata dal contenuto dei registri della *CPU*
- anche tale parte va salvata nelle strutture dati del *SO* quando il processo cessa l'esecuzione, così da poterla ripristinare quando il processo tornerà in esecuzione
- per rappresentare il contesto di un processo si usano due strutture dati:
  - un **descrittore di processo**, forse la struttura dati fondamentale del nucleo
  - l'indirizzo del descrittore costituisce un identificatore univoco del processo
  - e una **pila di sistema** operativo del processo (in breve chiamata **sPila**)
- ***si tenga ben presente che ciascun processo ha una sua sPila, diversa dagli altri***

Descrittore di Processo – File <linux/sched.h>

- questa struttura dati viene allocata dinamicamente nella memoria dinamica del nucleo ogniqualvolta viene creato un nuovo processo
- negli esempi i processi saranno spesso indicato con un nome:
  - si fa l'ipotesi che questo sia il nome di una variabile contenente un riferimento (puntatore) al descrittore di processo vero e proprio
  - per esempio si dirà: esistono due processi P e Q, intendendo che i simboli P e Q siano due variabili dichiarate nel modo seguente:

    ```
    struct task_struct * P
    struct task_struct * Q
    ```

    cioè come struct del linguaggio C, con numerosi campi contenuti
- pertanto la notazione `P→pid` indica il campo pid del descrittore del processo P

Struttura del Descrittore di Processo

```
struct task_struct {
   pid_t pid, tgid;
   volatile long state;   // -1 unrunnable, =0 runnable, >0 stopped
   void * stack;          // puntatore alla sPila del task
   struct thread_struct thread;  // struct per il contesto HW
   // variabili di vario tipo utilizzate per gestire lo scheduling
   // puntatore alle strutture utilizzate per gestire la memoria
   struct mm_struct * mm
   // variabili utilizzate per restituire lo stato di terminazione
   struct fs_struct * fs;         // filesystem information
   struct files_struct * files;  // open file information
   ...                           // altre informazioni ausilarie
} / * task_struct */
```

La Struttura per il Contesto *HW* (solo puntatori alla pila)

- struttura dipendente dallo hardware
- definizione diversa per ogni architettura

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

Accesso al Descrittore tramite un Kernel Module

si provi a inserire nel kernel module “axo_hello” la chiamata a questa 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);
   // continua a destra
   // 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);
} / * task_explore */
```

Risultato dell'Esecuzione di "axo_hello" con "task_explore"

- la stampa del risultato è fatta dal *SO* eseguendo il modulo «axo_hello» esteso con la funzione «task_explore»
- per questo motivo i messaggi hanno la numerazione dei messaggi di sistema (log di sistema)
- ovviamente la stampa del *PID* e del *TGID* può essere ottenuta anche con un programma applicativo che usa i servizi opportuni (*get_pid*, ecc)
- invece gli indirizzi della pila di sistema non potrebbero essere ottenuti da un normale programma; la loro interpretazione verrà discussa più avanti

```
[27663.251212] Inserito modulo Axo Hello.
[27663.251214] PID = 7424, TGID = 7424
[27663.251215] thread.sp =  0x FFFF 8800 5C64 5D68
[27663.251216] thread.sp0 = 0x FFFF 8800 5C64 6000
[27663.251217] ts->stack =  0x FFFF 8800 5C64 4000
[27663.251218] usersp =     0x 0000 7FFF 6DA9 8C78
[27663.260995] Rimosso modulo Axo Hello.
```