Sistemi Operativi, Secondo Modulo, Canale A–L e Teledidattica
Riassunto della lezione dell'11/03/2020

Igor Melatti

I processi

- Un sistema operativo Linux è composto, oltre che da files, da processi in esecuzione
  - volendola dire tutta: *Ogni risorsa di un sistema Linux è o un file o un processo*
- Ogni file *eseguibile* può diventare un processo in esecuzione
  - è sufficiente invocarlo, ad esempio con un comando sulla shell
  - ma anche cliccando 2 volte (o 1 volta, se si trova in un menu) sull'icona del file nell'interfaccia grafica
  - quasi tutti i comandi visti finora creano un processo
  - pochissime eccezioni, che riguardano tutte i cosiddetti *comandi built-in* delle shell
    - ad esempio, ricade in questa categoria il comando `cd`
    - analogamente, c'è anche il comando `echo [argomenti]`, che scrive su schermo tutto quello che c'è in `argomenti` (con qualche eccezione sui caratteri speciali; ci ritorneremo)
  - in questi casi, il tutto viene gestito dalla shell (ovvero dal processo in esecuzione che gestisce la shell) senza creare un ulteriore processo
  - ad esempio, per gestire `cd`, sarà sufficiente cambiare valore di una particolare variabile del programma della bash... non serve creare un processo per fare ciò
  - comunque, qualsiasi computazione è contenuta, in qualche modo, dentro ad un processo
- Quindi, un processo è un'istanza di un file eseguibile che sia effettivamente in esecuzione
  - quindi, se si esegue k volte lo stesso file, si creano k processi

  - non è necessario aspettare che un processo termini per lanciarne un altro: Linux è *multi-processo* (o *multitasking*)
  - se si esagera, le prestazioni del sistema risulteranno degradate

- Ogni processo è identificato da un numero, detto *PID* (*Process IDentifier*)
  - in un dato istante, non ci possono essere 2 processi con lo stesso PID
  - tuttavia, una volta che un processo è terminato, il suo PID viene "liberato", e potrebbe essere prima o poi riusato per un altro processo

- Ogni processo è rappresentato in memoria da:
  - 1 struttura dati mantenuta in RAM dal kernel e denominata *PCB* (*Process Control Block*); in tale struttura dati sono presenti i seguenti campi:

    PID: Process Identifier

    PPID: Parent Process Identifier

    Real UID: Real User Identifier

    Real GID: Real Group ID

    Effective UID: Effective User Identifier

    Effective GID: Effective Group ID

    Saved UID: Saved User Identifier

    Saved GID: Saved Group Identifier

    Current Working Directory: directory di lavoro corrente

    Umask: file mode creation mask

    Nice: priorità statica del processo
  - 6 aree di memoria (che possono essere all'occorrenza swappate su memoria virtuale, ovvero su disco), vedere anche Figura 1:

    Text Segment: le istruzioni da eseguire (in linguaggio macchina)

    Data Segment: i dati *statici* (ovvero, variabili globali e variabili locali `static`) inizializzati e alcune costanti di ambiente

    BSS: i dati *statici* non inizializzati (sta per *block started from symbol*); la distinzione dal segmento dati si fa per motivi di realizzazione hardware

    Heap: i dati *dinamici* (allocati con `malloc` e simili)

    Stack: le chiamate a funzioni, con i corrispondenti dati *dinamici* (variabili locali non `static`)

    Memory Mapping Segment: tutto ciò che riguarda librerie esterne dinamiche usate dal processo, nonché estensione dello heap in alcuni casi (system call `mmap`)
  - il PCB è uno per ogni processo; invece, alcuni processi potrebbero condividere (*sharing*) alcune delle 6 aree di memoria elencate sopra

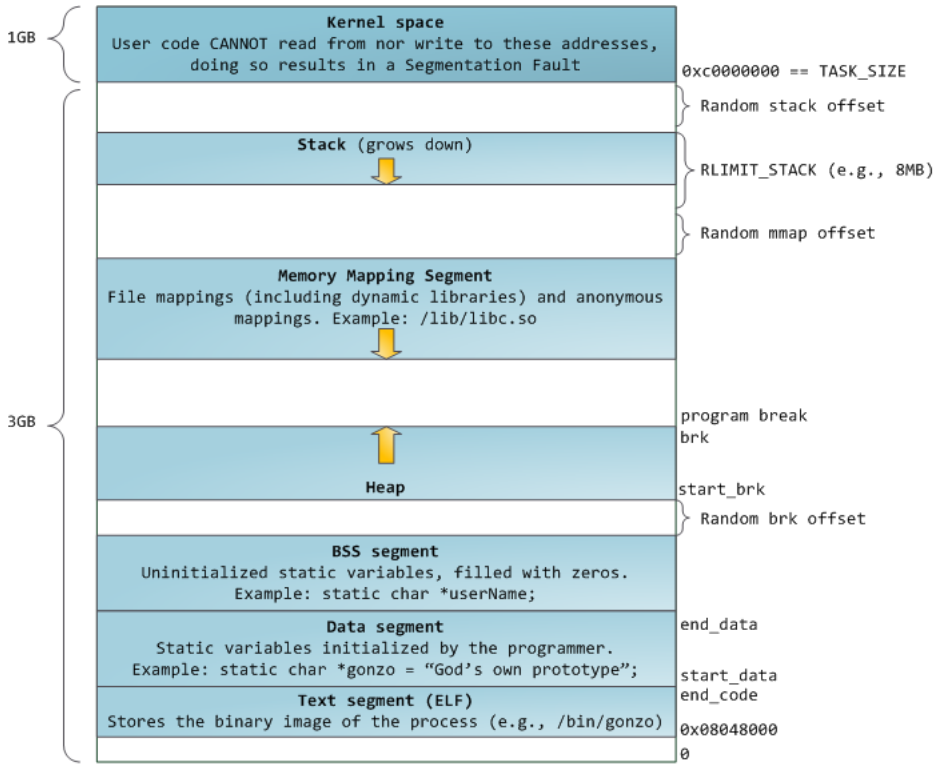

Figure 1: Processi in Linux

    - per esempio, se lancio 2 volte lo stesso programma, tipicamente il suo text segment viene caricato in memoria una volta sola
    - è possibile che due processi abbiano la stessa memoria, se si usano particolari accorgimenti in fase di programmazione
    - lo stack non è mai condiviso
  - Parentesi sui thread: da un punto di vista del PCB, per Linux ogni thread è un processo (*lightweight process*)
    - quindi, ogni thread di uno stesso processo ha un PID diverso
    - semplicemente, esistono macro interne al kernel che permettono facilmente di trovare tutti i PID dei thread di uno stesso processo
    - i thread di uno stesso processo condividono tutte le suddette aree di memoria, con l'eccezione dello stack, che invece è uno dedicato per ogni thread

- Ogni processo ha uno *stato* (ma sarebbe meglio parlare di modalità) preso tra i seguenti:

  Running (R): in esecuzione su un processore

  Runnable (R): non è in attesa di alcun evento, quindi può essere eseguito (ma non lo è perché lo scheduler ancora non lo ha (ri)selezionato)

  (Interruptible) Sleep (S): è in attesa di un qualche evento (ad esempio, lettura di blocchi dal disco), e non può quindi essere scelto dallo scheduler

  Zombie (Z): il processo è terminato e le sue 6 aree di memoria non sono più in memoria; tuttavia, il suo PCB viene ancora mantenuto dal kernel perché il processo padre non ha ancora richiesto il suo "exit status" (ritorneremo su questo punto)

  Stopped (T): caso particolare di *sleeping*: avendo ricevuto un segnale `STOP`, è in attesa di un segnale `CONT`

  Traced (t): in esecuzione di debug, oppure in generale in attesa di un segnale (altro caso particolare di *sleeping*; vedere più avanti)

  Uninterruptible sleep (D): come sleep, ma tipicamente sta facendo operazioni di IO su dischi lenti e non può neanche essere ucciso

- Negli esempi che seguono, si useranno due semplici comandi per creare processi: `sleep number[suffix]` e `yes [string]`
  - il primo si mette in pausa per `number` secondi (o ore, se `suffix` è `h`, o minuti, se `suffix` è `m`, o giorni, se `suffix` è `d`)
  - il secondo scrive all'infinito `y`, andando ogni volta a capo (oppure `string`, se viene data); per ora, verrà eseguito nel seguente modo: `yes niente > /dev/null`
  - serve ad evitare che generi output

- Uso del carattere `&` (*ampersand*): come lanciare un processo in *background*
  - non disponibile in tutte le shell, ma in bash sì
  - è strettamente collegato al concetto di *job*
  - per i nostri fini, un job è un comando che può essere eseguito in 2 modalità:

    foreground: praticamente tutti i casi visti finora, in cui non c'era il `&`. Il comando può leggere l'input da tastiera e scrivere su schermo; finché non termina, il prompt non viene restituito e non si possono sottomettere altri comandi (o job...) alla shell

    background: il caso di `geany` (usabile con tutti i programmi grafici...), in cui c'era il `&`. Il comando non può leggere l'input da tastiera ma può scrivere su schermo. Il prompt viene immediatamente restituito; mentre il job viene eseguito in background, si possono da subito dare altri comandi alla shell
      - nota: se lanciate in background un programma che deve leggere da tastiera, quello si blocca (cioè, va in stato stopped) non appena avviene la richiesta di input da tastiera
      - e fin qui, niente di strano, lo farebbe anche un programma in foreground
      - il problema è che il programma in foreground ha il terminale a sua disposizione, dal quale acquisire il suo input da tastiera
      - quello in background no, quindi resterà in stopped per sempre
      - a meno che non lo si riporti in foreground (provare ad esempio con il comando `bc`, che è un basic calculator)
      - il problema è risolvibile a monte (ovvero, a tempo di esecuzione del programma) usando le redirezioni, ci arriveremo
  - per ogni shell, ci può essere quindi un solo job in foreground, ma tanti in background (numerati con un id speciale, chiamato *job id*, progressivo da 1)
  - quando un job in background termina, l'utente viene avvisato (una volta terminato il primo comando che viene dato dopo la terminazione)
  - comando built-in `jobs [-l] [-p]`: mostra i job in background (ovviamente, solo della shell attuale); senza nessuna opzione, mostra numero del job e comando completo; con `-l` mostra anche il PID del processo corrispondente
  - una precisazione: in realtà, ogni job può contenere più comandi, concatenati in *pipelining*
  - in questo caso, `jobs -l` mostra, per ogni job, tutti i PID dei processi coinvolti

  - ci ritorneremo quando parleremo del pipelining
  - per mandare un job in background c'è anche una via alternativa: lo si lancia in foreground, poi si preme CTRL+z (vedere più sotto) e si scrive `bg`
  - il CTRL+z ha l'effetto di mettere il processo in stato *stopped*: potrà essere nuovamente rischedulato solo dopo che avrà ricevuto un segnale di continuazione, che è quello che fa `bg`
  - inutile fare `man bg`: è built-in di bash, quindi occorre fare `man bash` e cercare `bg`
  - da notare che si possono inviare molti comandi, premendo poi CTRL+z ogni volta, e senza scrivere poi `bg`
  - se a questo punto si vuole riportare in esecuzione in background uno qualsiasi dei job stoppati, come si fa?
  - occorre aver fatto caso al numeretto tra parentesi quadre che la bash ha ritornato dopo la pressione del CTRL+z
  - in alternativa, è lo stesso che viene mostrato dal comando `jobs`
  - a quel punto si può dare il comando `bg %`n, se n è il numero del job così determinato
  - in alternativa, c'è anche il comando `fg %`n, che prende un job in background e lo porta in foreground
  - `fg` può essere usato anche su un job stopped (ad esempio, dopo un CTRL+z); anche qui, il processo viene riportato in vita, ma questa volta in foreground
  - volendo complicarsi la vita, si possono identificare job anche con:
    * `%prefix`: dove `prefix` è la parte iniziale del comando del job desiderato;
    * `%+` oppure `%%`: l'ultimo job mandato
    * `%-`: il penultimo job mandato

- Comando `ps [opzioni] [pid...]`: mostra informazioni sui processi in esecuzione
  - qui le opzioni sono molte e variegate, alcune con significato simile se non uguale
  - il problema è che si mischiano opzioni "UNIX" (dash singolo), opzioni "BSD" (senza dash, comune anche ad altri comandi), e opzioni "GNU" (con doppio dash, non raggruppabili)
  - si possono mischiare le opzioni tra loro, ma il suggerimento è di non farlo (si rischiano confusione e comportamenti imprevisti)
  - senza nessun argomento, mostra solo i processi dell'utente attuale, che siano stati lanciati dalla shell corrente

- ovviamente, occorre che siano stati lanciati in background, altrimenti la shell non restituisce il prompt e quindi non si può dare questo comando (con l'eccezione dello stesso processo `ps`...)
- dei processi, sempre senza argomenti, mostra il PID (header: PID), il terminale da cui sono stati lanciati (header: TTY), il tempo di CPU (header: TIME), ed il comando usato (senza argomenti, header: CMD)
- per prepararsi ad un esempio, lanciare:

  ```
  sleep 60 &
  sleep 80 &
  yes niente > /dev/null &
  yes altroniente > /dev/null &
  ```

- provare a dare il comando `jobs -l`
- provare prima con solo ps; aprire un'altra finestra e scrivere nuovamente ps
- nel seguito, vengono esposte le più importanti opzioni di `ps`, nel formato "UNIX":
  - `-e`: mostra tutti i processi in esecuzione, di qualsiasi utente essi siano
    - dati su qualsiasi shell, o non dati da shell
    - infatti, i processi potrebbero essere stati creati come figli da parte di altri processi
    - e potrebbero essere figli dei processi "nati" durante il boot, o lanciati tramite interfaccia grafica
    - da notare che anche i processi lanciati da shell sono "figli" di un altro processo: la shell stessa
    - l'unica eccezione al fatto che "tutti i processi nascono da altri processi" è il processo 0, che viene creato accendendo la macchina (e termina dopo aver creato il processo 1, sempre presente mentre Linux è in esecuzione)
  - `-u` {`utente,`}: mostra tutti i processi dei soli utenti specificati
  - `-p` {`pid,`}: mostra tutti i processi aventi i pid specificati
    - si può ottenere lo stesso effetto dando tali pid come argomenti (ovviamente, separati da spazi e non da virgole)
  - `-f`: aggiunge alle informazioni mostrate anche (vengono qui riportati con gli header corrispondenti):
    - PPID (*parent ppid*: pid del processo che ha creato questo processo); per i processi lanciati da una certa bash, sono tutti figli della bash...
    - C: parte intera della percentuale di uso della CPU
    - STIME (o START): l'ora in cui è stato fatto invocato il comando, oppure la data se è stato fatto partire da più di un giorno

    * TIME: tempo di CPU usato finora
    * CMD: comando con argomenti

`-l`: aggiunge alle informazioni mostrate anche (vengono qui riportati con gli header corrispondenti):

    * F: flags associati al processo; può valere: 0, 1, 4, 5
        · 1: il processo è stato "forkato", ma ancora non eseguito
        · 4: ha usato privilegi da superutente (da notare che li deve aver usati, i privilegi, non solo avere il potere di usarli)
        · 5: entrambi i precedenti
        · 0: nessuno dei precedenti
        · comunque poco utile: l'opzione `-y`, se abbinata a `-l`, lo toglie
    * S: stato (modalità) del processo in una sola lettera, vedere sopra (le lettere corrispondenti sono tra parentesi)
    * UID: utente che ha lanciato il processo (si tratta dell'utente effettivo, quindi in caso di setuid non è necessariamente chi ha dato il comando; quest'ultima informazione è nel RUID, o *real UID*)
    * PID
    * PPID
    * C
    * PRI: attuale priorità del processo (più il numero è alto, minore è la priorità)
    * NI: valore di nice, da aggiungere alla priorità (vedere più avanti)
    * ADDR: indirizzo in memoria del processo, ma è mostrato (senza valore) solo per compatibilità all'indietro (infatti, c'è l'opzione `-y` che, combinata con `-l`, lo toglie)
    * SZ: dimensione totale attuale del processo (tutti i 6 campi detti sopra), ma in numero di pagine
        · da non confondere con RSS (*resident set size*), mostrata al posto di ADDR se c'è anche l'opzione `-y`: questa volta viene mostrata la dimensione attuale del processo (in kB), ma limitatamente alla parte in memoria principale
        · quindi, sono escluse dal conteggio di RSS le pagine swappate su memoria secondaria per via del meccanismo della memoria virtuale
        · inoltre, c'è il campo VSZ (per mostrarlo, vedere più sotto): mostra SZ in kB
    * WCHAN: se il processo è in attesa di un qualche segnale o comunque in sleep, qui c'è la funzione del kernel all'interno della quale si è fermato

    * TTY
    * TIME
    * CMD
  - `-o {field,}`: per scegliere quali campi visualizzare; nel man ci sono gli acronimi; ad esempio, per visualizzare il campo VSZ, occorre dare `-o vsz` oppure `-o vsize`
    * attenzione, non può essere data insieme a `-l`
    * l'opzione `-o vsz` mostra *solo* VSZ...
  - **esercizio:** copiare `/bin/sleep` nella directory corrente, cambiargli proprietario e gruppo in `root` e mettergli il setuid (occorre `sudo`...); dopodoché, lanciarlo in background e farsi stampare sia l'utente reale che quello effettivo del processo

- Comando `top [-b] [-n num] [-p {pid,}]`: un `ps` interattivo e che fa refresh ogni pochi secondi
  - con l'opzione `-b` non accetta più comandi interattivi, ma continua a fare refresh ogni pochi secondi
  - con l'opzione `-n num` fa solo `num` refresh
  - l'opzione `-p` è come quella di `ps`
  - una volta aperto in modo interattivo, basta premere `?` per avere la lista dei comandi accettati (tipicamente, sono singole lettere)
  - **esercizio:** fare in modo che `ps` abbia lo stesso output di `top -b -n1`

- Mandare un segnale ad un processo o ad un job: comando `kill [-l [signal]] [-signal] [pid...]`
  - si chiama kill e *può* essere usato per "uccidere" (terminare) un processo ma *non è* l'unico motivo per cui lo si usa
  - con l'opzione `-l` mostra quali segnali possono essere usati
  - viene usato per mandare il segnale `signal` (che può essere o un numero o il nome del segnale, anche senza SIG) ai processi dati come argomento (o a tutti i processi, se l'argomento non c'è)
  - i segnali verranno presi in considerazione solo se il *real user* del processo ricevente è lo stesso del processo che invia il segnale (oppure, se lo invia un processo di un superuser)
  - un processo che riceve un segnale fa o un'azione predefinita (vedere `man 7 signal`) o un'azione personalizzata (su questo torneremo in un'altra lezione)
  - per ora ci limiteremo ai segnali per la sospensione di processi (SIGSTOP e SIGTSTP), per la continuazione di processi stoppati (SIGCONT) e per la terminazione di processi in esecuzione (SIGKILL e SIGINT)

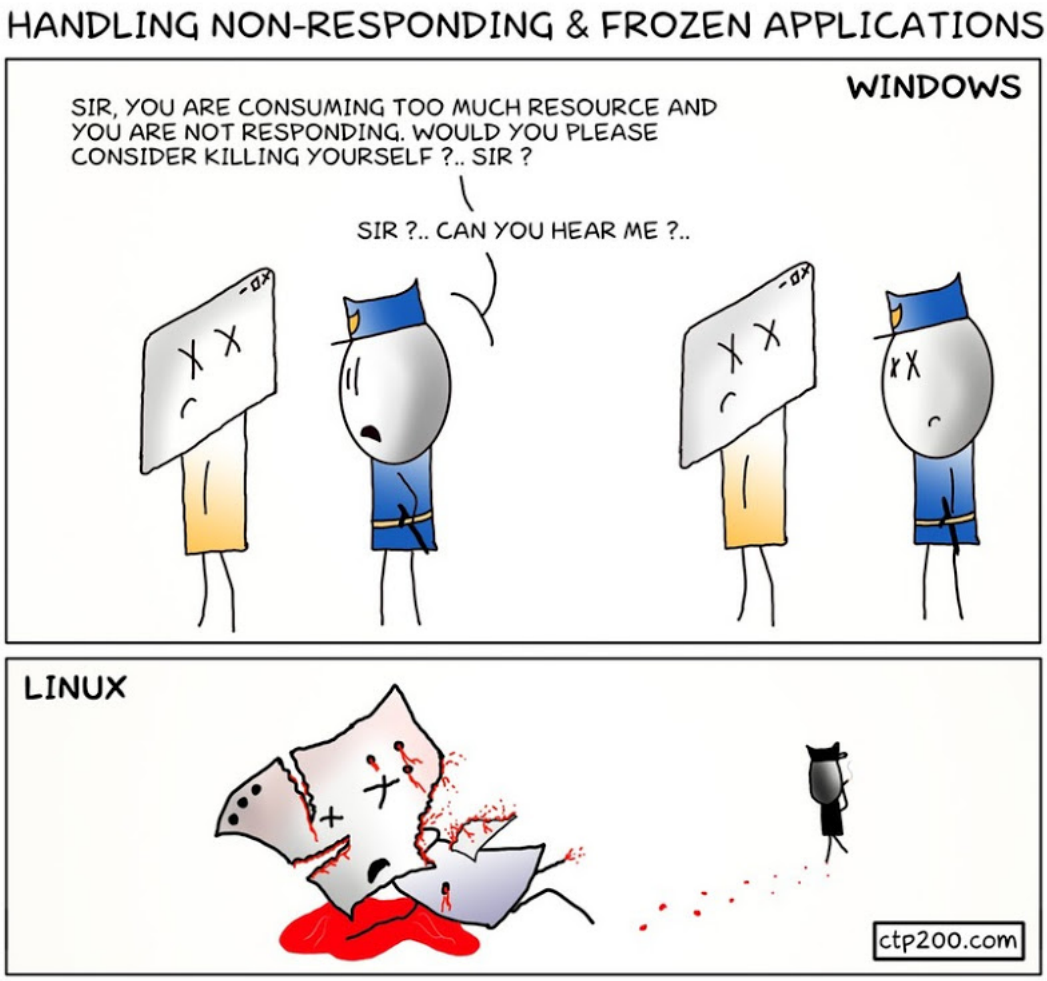

Figure 2: Killare un processo in Linux e Windows

  - per mandare SIGSTOP si può scrivere o `kill -SIGSTOP pid` o `kill -STOP pid` o `kill -19 pid`
  - notare che se si stoppa un comando `sleep`, lui continua a "contare" i secondi (in realtà, non è proprio così, ma l'effetto è quello); se lo si fa ripartire o termina subito (se il numero di secondi è già passato), oppure resta in esecuzione solo per il numero di secondi rimasti
  - alcuni segnali possono essere mandati ad un comando (che sia però un job in foreground nella shell attuale) usando la tastiera: CTRL+z manda SIGTSTP, CTRL+c manda SIGINT
  - il comando `bg` manda SIGCONT al job indicato (l'ultimo se non ci sono argomenti)
  - si possono lanciare segnali a job usando la stessa notazione con il carattere `%` vista per `bg` e `fg`
  - **esercizio:** lanciare un processo in foreground, poi (usando un'altra shell...) simulare la pressione di CTRL+z e di CTRL+c, nonché il comando `bg` (per sapere il PID di un processo in esecuzione su un'altra shell, si può usare il nome del comando, vedere l'opzione `-C` di `ps`)
  - **esercizio:** verificare di avere almeno un GB di spazio, e creare con `dd` un file di 1 GB (tutto di zeri). Subito dopo averlo lanciato, mandare al processo di `dd` il segnale USR1. Cosa succede?

- Comandi `nice [-n num] [comando]` e `renice priority {pid}`
  - il *niceness* può essere pensato come una addizione sulla priorità: se positivo, ne aumenta il valore (quindi la priorità decresce), altrimenti ne diminuisce il valore (la priorità cresce)
  - va da -19 a +20, con default a 0
  - `nice` senza opzioni dice quant'è il niceness di partenza (zero)
  - altrimenti, lancia `comando` (che potrebbe avere delle sue opzioni, più o meno come con `sudo`) con niceness `num` (0 se non dato)
  - `renice` invece interviene su processi già in esecuzione
  - infatti, richiede dei PID (ci sarebbero anche altre opzioni per fargli avere effetti sui processi di un dato utente/gruppo, ma soprassediamo)
  - **esercizio:** quindi, basterebbe mettere `-n num` negativo per avere una migliore esecuzione per i propri processi! provare...

- Comando `strace [-p pid] [comando]`
  - se `comando` è dato, lo lancia e visualizza tutte le sue system call
  - altrimenti, visualizza le system call del processo con PID `pid`