Materiale di approfondimento

- Lectures on UNIX
- understanding processes on Linux

Esercizi

- overthewire Bandit

Lab = matteo.zoia@unimi.it | andrea.monzani@unimi.it

Intro Linux

Un **sistema di calcolo può essere suddiviso per strati** e i componenti possono essere classificati per funzione logica:

- abbiamo da una parte le **applicazioni**, le applicazioni usano il sistema operativo per funzionare.
- **sistema operativo** a sua volta si appoggia sul system software che è un componente che fa parte del kernel.
- **system software** è un programma che ha il compito di gestire le risorse e fornisce servizi di base ai software di livello superiore.

sys user | app software | OS | sys software (kernel) | hardware

Il sistema operativo prende il controllo della macchina nel momento in cui accendiamo il pc, è il programma con cui l'utente interagisce.

I **compiti** dell'OS sono :

- **garantire confidenzialità, affidabilità, disponibilità** delle informazioni che l'utente immette nel sistema.
- **gestire** esclusivamente le **risorse** (cpu, mem, periferiche) (è l'unico programma abilitato a gestire le risorse).
- deve **garantire la ottimalità dell'uso delle risorse**.

le applicazioni non possono dialogare direttamente con hardware, devono fare un `syscall` mediante OS.

Gestore: è entità che decide quanto memoria dare, quanto togliere per dare ad un'altra applicazione, sicurezza del sistema. Inoltre, deve garantire confidenzialità, integrità, disponibilità.

input/output management, command interpreter, resource management, file, memo, security.

Un utente può interagire con l'OS programmando oppure con il shell.

Insieme ad un OS abbiamo la libreria delle `system calls`, consentono al programmatore di usare le risorse del sistema.

I sistemi operativi si distinguono per `syscall`, dalle utilità di sistema, interfaccia utente, politiche di scheduling delle risorse.

utenti
app (user app
OS (syscall process manager, mem, file sys, IO services)
hardware

Linux - Introduction

È un derivato di UNIX. UNIX nasce alla fine di 1960. deriva da un progetto **Multics** (multiplexed information and computing sys). Ken Thomposon UNIX **uniplexed information and computing system**. Riscrive in c, quello che favorisce la diffusione è che era **open source**. Quindi la gente si interessarono di più per testare ecc.

Nel 78 UNIX si divide, AT&T compra kernel di UNIX e diventa un sistema proprietario "sys 5" e la parte open source BSD Berkley System Distribution.

Nel 1991 tutto converge in Linux sviluppato da Linus Travolds.

web-vm Linux

shell = interprete del comandi read cmd -> interpreta -> execute -> display -> read

un commando è formato

```
cmd -opts targets
-------------------
dmseg | more
id
who
who am i
last
finger
w
yes please
echo {con,pre}{sent,fer}{s,ed}
man "automatic door"
who can tell me why i got divorced
lost
cal 2000
cal 9 1752 // è quando stato adottato il calendario gregoriano
bc -l
echo 5+4 | bc -l
```

Quando eseguo un cmd l'OS crea un processo.

Come viene creato un processo in Linux/UNIX

Quando un utente esegue un programma (sequenza di istruzioni) l'OS genera un processo.

Processo **è l'insieme di attività che l'OS svolge per poter eseguire le istruzioni contenute all'interno di un programma**. Le risorse (sono private al processo) che un processo ha sono: memoria centrale, cpu, risorse di I/O.

Per mandare in esecuzione un programma l'OS utilizza la nozione di processo e prima di mandare in esecuzione l'OS controlla le risorse di cui ha bisogno il programma. (Es. quanta memoria ho a disposizione, ecc.)

Ad ogni cmd di shell corrisponde un programma che stanno nella directory `user/bin`. Un processo può essere generato anche dal programmatore.

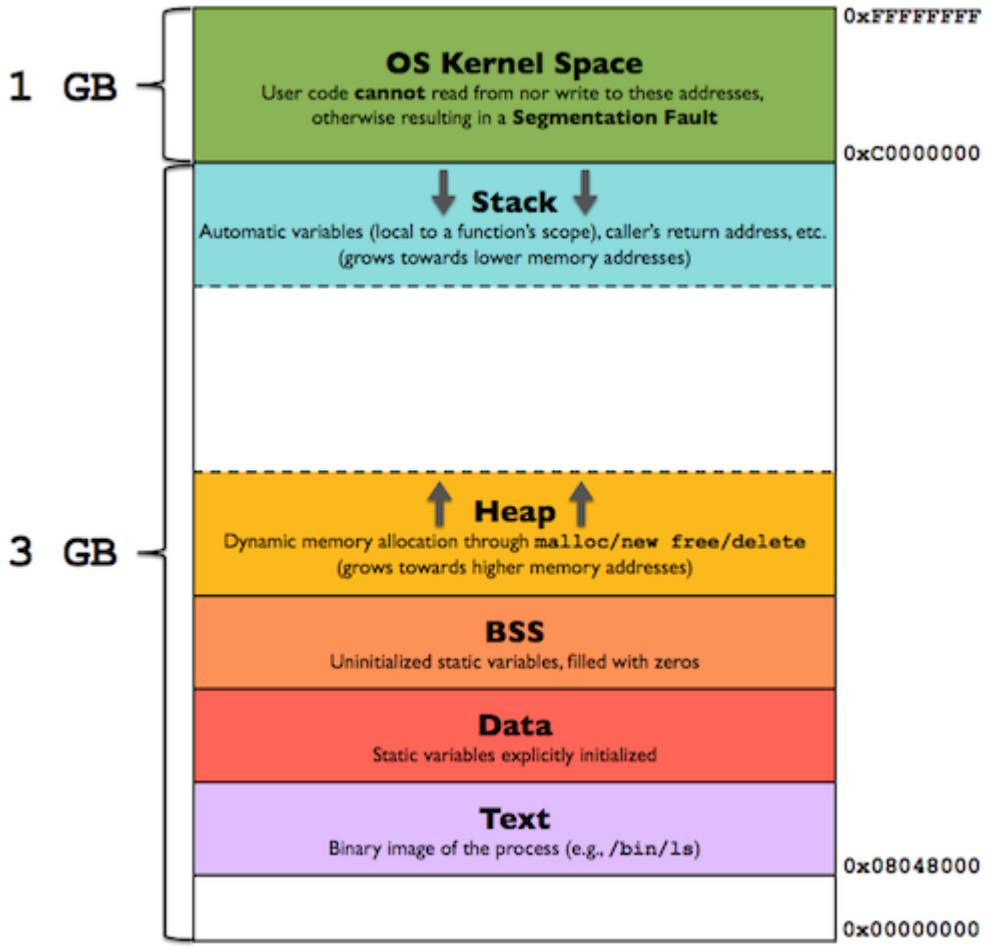
`syscall` che si utilizza per generare un processo è `fork()`.

Come avviene la trasformazione da un programma ad un processo?

Si **porta in memoria centrale** il programma (la sequenza di istruzioni) tipicamente dal disco fisso -> viene **riservato uno spazio di memoria** per memorizzare le variabili e viene allocato un altro spazio di memoria per le variabili dinamiche (sono le variabili che vengono allocate per la durata dell'esecuzione di un programma, cioè, restano in memoria solo per la durata che serve (sullo stack)) (variabili statici (costanti)) -> a questo punto, viene creato un process control block **PCB**

PCB è una struttura dati che viene usata dall'OS per tenere traccia delle attività che viene svolto dal processo:

- l'indirizzo dello stack di quel processo,
- l'indirizzo dove ha caricato le istruzioni del processo.



Quando si hanno più processi contemporaneamente si fa *time sharing* per condividere il processore. La tecnica che si usa tipicamente è il *round robin*.

Mono programmato: un unico processo in memoria, viene eseguito un processo alla volta.

Multi programmati: sono presenti più processi in memoria.

Content Switch

È la tecnica che viene utilizzato per liberare il processore e far spazio ad un nuovo processo consiste nel congelare lo stato di un processo e metterlo da parte (salvati in memoria/disco fisso) per poi ricaricarlo nel processore per continuare. Il **contesto di un processo** (PCB) ha un pid, cpu state, processor, memory, program counter, ecc.

Il context switch può essere eseguito, quando scade il quanto di tempo, quando viene generato un interrupt di I/O, quando si verifica un segmentation fault oppure quando si verificano delle eccezioni durante il funzionamento di un programma.

Context switch è un *interrupt driven operation*.

Ogni processo di UNIX ha un genitore/un proprietario.

Interrupt

`clock interrupt`

L'interrupt è un segnale hardware che interrompe il ciclo fetch-decode-execute del processore. Sul processore ce un **pin dedicato per ricevere il segnale interrupt** (elettrico-hardware), dopo l'execute ce un `if` che controlla se ce un interrupt, che va interrompere l'esecuzione del programma in corso e va ad eseguire un altro programma che è in una posizione ben definita della memoria.

`if interrupt go to`

In assembly il commando `INT` ha un particolare indirizzo dell'OS che contiene un routine di gestione dell'interrupt, che provvede di salvare lo stato del processo in corso (fare content switch) e manda in esecuzione il resto dell'OS.

Quindi, l'interrupt è un meccanismo hardware, è la modalità con cui le periferiche interagiscono con l'OS. Esiste anche una modalità software, es. quando si genera un segmentation fault (il processo viene interrotto, chiama l'OS che cerca di capire cosa sta succedendo, ammazza il processo, libera la memoria, l'eventuale errore lo mette in un dump della memoria e fa ripartire il sistema), l'interrupt software = **exception** (errori dell'interno di un programma).

Interrupt è un bit (piedino del cpu), le cause dell'interrupt (es. mouse, disco fisso, ecc.) quindi ognuno ha un codice che identifica il suo interrupt, questo codice è chiamato **vettore dell'interrupt** viene passato al sistema in una fase successiva alla segnalazione dell'interrupt. Il codice viene usato per accedere all'array dell'OS che contiene gli indirizzi corrispondenti alla gestione di ogni tipo di interrupt.

I routine sono `interrupt handler`, `exception handler`.

CPU is executing instruction 100 (programma in esecuzione) -> p1 (periferica) (un indirizzo memory mapped) ha un vettore di interrupt 16 -> p1 manda un segnale di interrupt (quando cpu sta eseguendo l'istruzione riga 100) -> cpu salva il contesto -> esegue il codice handling interrupt e chiede a p1 di mandare il vettore di interrupt -> p1 manda il vettore (16) -> cpu salta all'indirizzo al interrupt handler di 16 -> esegue l'interrupt handler -> alla fine del codice di gestione dell'interrupt ce un commando assembly `RETI` return from interrupt, che permette di ripristinare l'esecuzione del programma principale.

Questo ci permettere avere un pseudo-multitasking.

Cosa succede durante il context-switching? viene salvato una serie di informazioni del programma in esecuzione (cosa salva dipende dall'OS) quindi un l'insieme di informazioni che consente all'OS di riprendere l'attività dal punto in cui è stato interrotto. Una volta salvato, viene eseguito l'interrupt handler e poi viene eseguito schedular.

`syscall` sono dei meccanismi che si appoggiano su interrupt. Sono in una libreria all'interno del kernel che contiene degli indirizzi di memoria. Quando viene generato un syscall blocca il processo in esecuzione e salta ad eseguire la syscall.

Syscall `fork()` permette di creare un processo, crea una copia fisica del processo genitore. unica cosa che cambia è il `PID` **process identifier**. Quindi, duplica il processo che viene chiamata.

`fork()` restituisce `-1` in caso di errore, `0` al processo figlio, `PID` al genitore (del figlio creato).

Il processo figlio eredita anche il program counter, quindi il child parte da dove era arrivato il genitore.

```
// killa n-2 genitori rimane solo ultimo child
pid_t childpid 0
for(i = 1; i < n; i++)
    if(fork() != 0) kill

// genera un processo child e lo ammazza
pid_t childpid 0
for(i = 1; i < n; i++)
    if(fork() == 0) kill
```

Quando un processo termina vengono chiusi tutti i file, temp file vengono cancellati, tutte le risorse dei processi child vengono deallocati, file descriptor, memory, traffic light, ecc...

Alla chiusura del processo child PCB del processo resta e sarà rimosso dal genitore.

In UNIX ogni processo ha un parent processo. Al parent processo viene mandato un **segnale alla chiusura** del child tramite il metodo `wait()`.

Cosa succede a un processo quando il processo si dimentica di fare la terminazione di child (senza fare wait)? Si creano **processi zombie** (processi fantasma) che sono dei processi che non hanno genitori. Questi processi vengono rimossi dall'OS periodicamente.

Linux process commands

tutti processi hanno un pid, pid = 1 processo init creato quando OS parte. da questo viene creato tutti pcb facendo fork sul processo init. procs possono operare in foreground `fg` / background `bg` (non interagiscono con utente)

`&` permette di avviare un programma in `bg code &`

processi possono essere sospesi con `ctrl+z`

`ps` currently running processes on this shell

`ps -fae` tutti processi del sistema

`kill pid` killa il processo in modo gracefull

`kill -9` pid killa forzatamente

`|` **pipe concatena** più comandi, output di un cmd viene mandato in input del prossimo command nella lista

`cmd1 | cmd2 | cmd3`

`ls -la | more` // visualizza 10 righe alla volta

`cat file.txt | grep "mammals"` // cat visualizza il contenuto del file e grep filtra le sotto stringhe che contengono la parola mammals.

I/O redirection

ogni processo ha tre file in UNIX: `std input` (0), `std output` (1), `std error` (2) hanno una postazione fissa

`>` per ridirigere output (sovrascrive il file)

`<` per ridirigere input

`>>` appende input in fondo al file

`1>` output al `std output`

`2>` output al `std error`

`cat < output > output` prima cosa che fa è creare il `output` e poi lo legge. quindi legge un file vuoto.