Quinta Esercitazione

Sommario

- Gestione dei processi
  - informazioni associate ai processi
  - identificatore di processo
  - real/effective UID (GID)
- Priorità dei processi
  - priorità assegnata dal SO
  - livello di nice
    - ⇨ comandi `nice` e `renice`
- Comandi per la gestione dei processi
  - `ps`: output e opzioni principali
  - `top`: output e comandi interattivi
- Segnali
  - descrizione dei segnali
    - ⇨ `SIGHUP, SIGTERM, SIGKILL, SIGSTOP`
    - ⇨ segnali prodotti da tastiera
    - ⇨ segnali invocati da processi
  - comandi per l'invio di segnali
    - ⇨ `kill, killall` e cenni a `nohup`
- Shell e processi
  - esecuzione in background e job
  - comandi `fg, bg` e `stop`
  - invio segnali da tastiera (^C, ^D)

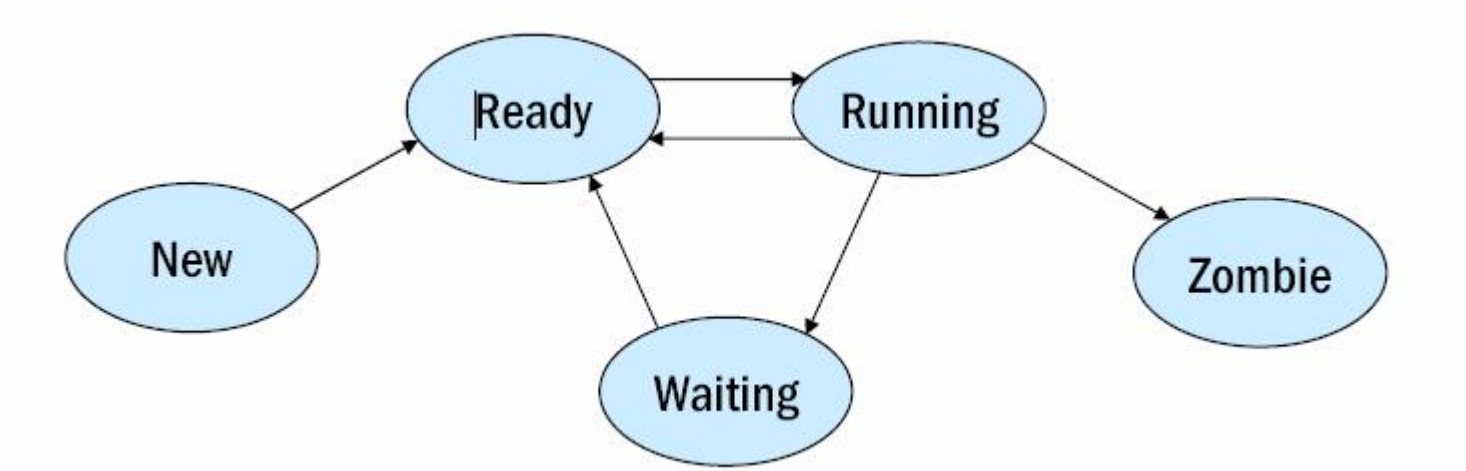
Gestione dei Processi

Processo

- Processo : programma singolo nel momento in cui viene eseguito
- I processi di dividono in:
  - Utente
  - Sistema

Stato di un processo

- Creazione di un processo
  - tramite la chiamata di sistema `fork()`
  - soltanto da un processo già attivo percio' si crea una gerarchia di processi (albero)
- Terminazione di un processo
  - tramite la chiamata di sistema `exit()`
  - "zombie": processo che ha completato la sua esecuzione ma che compare ancora nella tabella dei processi (exit status)



init

- Il processo `init` `(/sbin/init)` e' la radice dell'albero
  - viene creato dal kernel al termine della procedura di bootstrap
  - effettua tutta una serie di azioni per portare il sistema in un certo stato
  - PID = 1
  - Non ha padre
- Quando un processo termina i suoi figli diventano figli di `init`

Identificazione dei processi

- PID: identificatore univoco di processo
- PGID: identificatore di gruppo-processi cioe' il gruppo di processi a cui appartiene
- UID: identificatore di utente
- GID: identificatore di gruppo-utenti
- PPID: PID del processo da cui e' stato generato

PID

- Numero a 16 bit (di tipo `pid_t`) assegnato sequenzialmente dal kernel ogni volta che un nuovo processo è creato
- >=0
- Univoco
- Viene reciclato quando termina
- =0 scheduler
- =1 `init`
- =2 pagedaemon

ID e privilegi

- UID e GID determinano i privilegi, ovvero quali system call il processo ha il diritto di invocare e su quali risorse

Real ID

- RUID e RGID: UID e GID dell'utente che ha mandato in esecuzione il processo
  - Generalmente i valori del *real user-id* e del *real group-id* non cambiano per tutta la sessione di login
  - solo il superuser ha il potere di cambiarli

Effective ID

- EUID e EGID: UID e GID dell'utente e del gruppo che il kernel considera per determinare i privilegi per l'accesso ai file
  - potrebbero non coincidere con RUID e RGID nel caso in cui il file eseguibile ha il *set-user-id* (SUID) o il *set-group-id* (SGID) bit attivo
  - Possono variare durante l'esecuzione del processo

Meccanismo SUID/SGID

- Ogni file ha un *owner* e un *group owner:* se il file di un programma ha attivo il bit dei permessi *SUID* (*SGID*), allora, quando viene invocato con una chiamata `exec`, l'EUID (EGID) diventerà quello dell'*owner* (*group owner*) del file e non quello dell'utente che lo ha lanciato.
  - il bit SUID assegna all'EUID l'UID del proprietario del file
  - il bit SGID assegna all'EGID il GID del proprietario del file

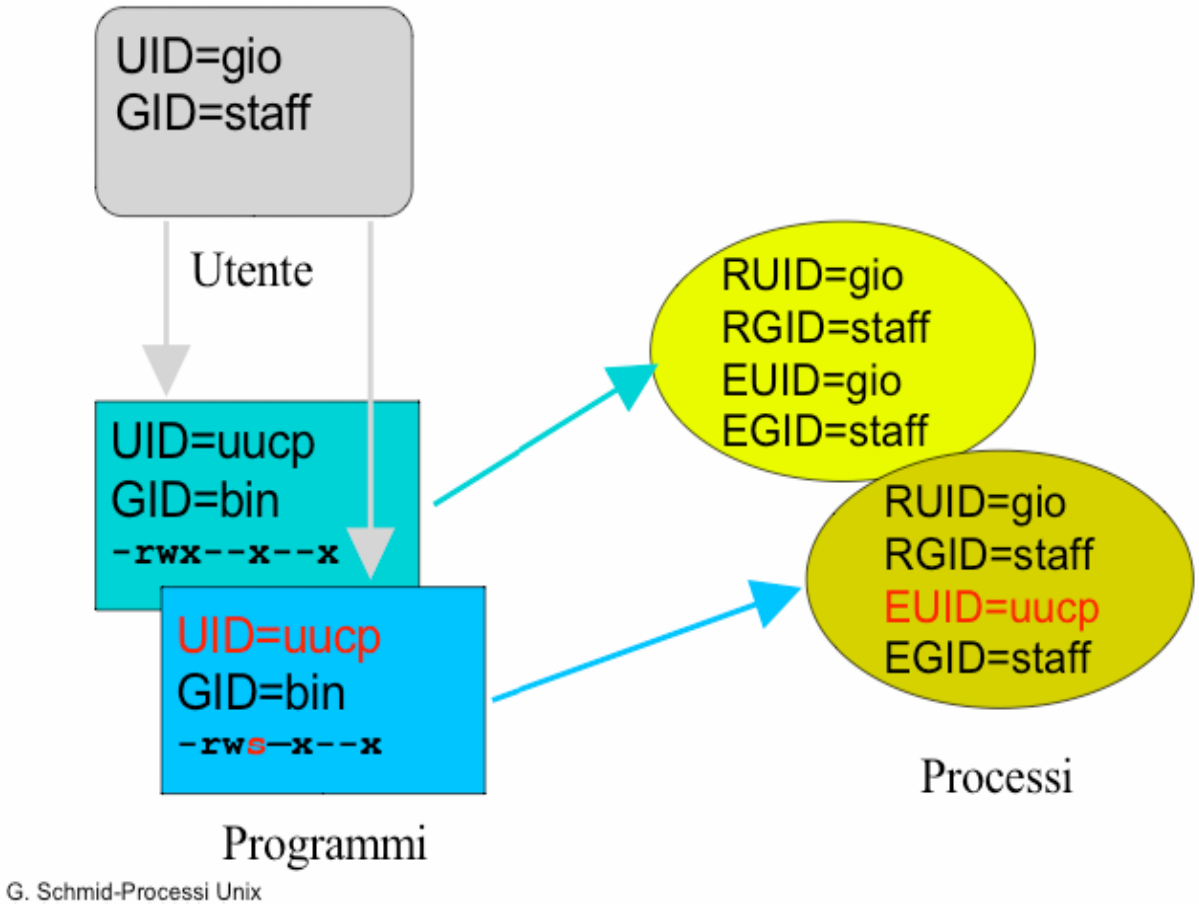
Set

- `int setuid(uid_t uid);`
- `int setgid(gid_t gid);`
- Cambiano real user/group ID ed effective user/group ID
- Esistono delle regole per permettere al programma di cambiare questi ID:
  - se il processo ha privilegi da superutente, la funzione `setuid` cambia RUID e EUID con `uid`
  - se il processo non ha privilegi da superutente e `uid` è uguale a RUID allora viene cambiato EUID
  - se nessuna di queste condizioni è vera, viene ritornato un errore ed `errno` è settato uguale a EPERM
- Per quanto riguarda `setgid`, le regole sono del tutto simili

Utilizzo di EUID/EGID

- **Se un file di programma appartiene al superuser ed ha il bit SUID attivo, l'utente che lancia il programma ottiene i privilegi del superuser durante l'esecuzione del programma stesso**
- **Un caso tipico è l'eseguibile `passwd` per cambiare la propria password:**

```
$ ls -l /usr/bin/passwd
-r-s--x--x 1 root root 19336 Sep 7 2004 /usr/bin/passwd
```


UID=gio
GID=staff
Utente
UID=uucp
GID=bin
-rwx--x--x
UID=uucp
GID=bin
-rws--x--x
Programmi
RUID=gio
RGID=staff
EUID=gio
EGID=staff
RUID=gio
RGID=staff
EUID=uucp
EGID=staff
Processi
G. Schmid-Processi Unix

Get

- `pid_t getpid(void)` : **restituisce il PID**
- `pid_t getppid(void)` : **restituisce il PPID**
- `uid_t getuid(void)` : **restituisce il RUID**
- `uid_t geteuid(void)` : **restituisce l'EUID**
- `gid_t getgid(void)` : **restituisce il RGID**
- `gid_t getegid(void)` : **restituisce l' EGID**
- **Non possono fallire**

Esempio

```
#include <unistd.h>
main(){
   uid_t uid, euid, newuid;
   gid_t gid, egid, newgid;
   int status;
   uid = getuid();
   euid = geteuid();
   gid = getgid();
   egid = getegid();
   printf(“real uid: %d, effective uid:
   %d\n”,(int)uid,(int)euid);
   printf(“real gid: %d, effective gid:
   %d\n”,(int)gid,(int)egid);
   if ((status = setuid(newuid))==0) /*cambio effective uid*/
       printf(“nuovo effective uid: %d\n”, (int)newuid);
   if ((status = setgid(newgid))==0) /*cambio effective gid*/
       printf(“nuovo effective gid: %d\n”, (int)newgid);
}
```

Priorità dei Processi

Priorità dei processi

- Lo "scheduler" UNIX (una parte del kernel) organizza l'esecuzione dei processi in base ad un certo livello di priorità.
- Tale priorità viene calcolata dal sistema in base a:
  - tipo di processo
  - comportamento del processo
  - variabile NICE.
    - ⇨ È da notare che non esiste tale variabile a livello di shell

```
ps -l
```

Valori di NICE

- Priorità:
  - 0 e 39
  - -20 e 19
  - **Più alta è la priorità più lentamente viene eseguito il processo.**
- NICE viene sommata nel calcolo della priorità.
- L'utente (normale o super-user) non può intervenire sulla priorità ma solo sul NICE

nice

- Quando viene creato un nuovo processo, alla variabile NICE viene assegnato un valore intermedio (20 o 0)
- L'utente può richiedere una variazione con il comando `nice`.
- L'utente può solo aumentare il NICE value.
- Non e' possibile ridurre un NICE value nemmeno per riportarlo al valore precedente da cui l'utente stesso lo ha elevato.

Super user e NICE

- Il super user può invece operare nell'intero campo NICE e su tutti i processi di sistema
- Ovviamente saranno necessarie opportune precauzioni soprattutto per i valori fortemente negativi.

Effetti di `nice`

- Se la variabile NICE aumenta il suo valore allora il processo esegue più lentamente alleggerendo il sistema.
- Un uso appropriato e' quello di lanciare i programmi in background con un NICE elevato.

renice

- Permette di variare il livello di NICE una volta lanciato il processo
- Un utente può applicare il `renice` solo ai propri processi.
- Il comando `renice` accetta opzioni che consentono di specificare i processi non solo per PID ma anche per utente o gruppo di processi.
- SYNOPSIS

```
renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]
```

Esempio

- **Lanciamo il comando `nroff` in background**

```
$ nroff documento > out &
```

- **Guardiamo il valore di NICE**

```
$ ps –xl
```

- **Cambiamo il livello di NICE**

```
$ renice valore PID
```

- **Verifichiamo il cambiamento**

Comandi per la gestione dei Processi

ps (1 di 3)

- `ps` : visualizza un elenco di processi in corso di esecuzione

PID	TT	STAT	TIME	COMMAND
32798	p0	Ss	0:00.03	-bash (bash)
32925	p0	R+	0:00.00	ps

  - PID : numero del processo
  - TT : terminale
  - STAT : stato del processo
    - ⇨ R : running
    - ⇨ I : bloccato < 20s
    - ⇨ S : bloccato > 20s
    - ⇨ D : pausa non interrompibile
    - ⇨ T : sospeso
    - ⇨ Z : zombie
    - ⇨ W : in memoria virtuale
    - ⇨ N : NICE > 0

Opzioni di ps (2 di 3)

- x
  - visualizza anche i processi che non provengono da terminali
- u
  - indica in particolare l'utente a cui appartiene ogni processo
- a
  - visualizza i processi di tutti gli utenti
- o
  - mostra i campi elencati di seguito (separati da virgole) più quelli di default
- o
  - precisa *solo* le colonne da mostrare (separate da virgole)
- U *nome_utente*
  - mostra i processi di *nome_utente*

ps (3 di 3)

- ps aux : **indica tutti i processi in esecuzione**

USER	PID	%CPU	%MEM	VSZ	RSS	TT	STAT	STARTED	TIME	COMMAND
vanessa	32929	0.0	0.1	1484	944	p0	R+	6:40PM	0:00.00	ps aux

- VSZ : memoria virtuale usata
- RSS : memoria fisica usata
  - VSZ >= RSS
- TT : terminale virtuale a cui è associato
  - TT = ?? Processi non associati a nessun terminale. Sono i DEMONI, cioè processi di sistema
- STARTED : orario di partenza del processo
- TIME : tempo in cui ha usato la CPU
- COMMAND : percorso che ha creato il processo

top (1 di 3)

- `top` : visione dinamica dei processi
  - load avarage : utilizzo della CPU nell'ultimo minuto, ultimi 5 e ultimi 15
  - CPU states
    - ⇨ user : usata dagli utenti
    - ⇨ system : usata dalle system call
    - ⇨ interrupt : carico dato dalle interruzioni
    - ⇨ nice : cresce quando cambio NICE
    - ⇨ idle : usata dalla dummy()
  - Mem
    - ⇨ Active : allocata ai processi
    - ⇨ Inact : aspettano di essere rieseguiti
    - ⇨ Wired : zona riservata dal kernel

top (2 di 3)

- PID
- USERNAME
- THR : thread
- PRI : priorità
- NICE
- SIZE : dimensione in memoria virtuale
- RES : resident
- STATE
- TIME
- WCPU : media veloce, finestra temporale stretta
- COMMAND

top (3 di 3)

- **Comandi interattivi**
  - h : **help**
  - r : **renice, chiede il nuovo livello**
  - s : **intervallo di aggiornamento**
  - k : **segnale**
  - n : **quantità di processi da visualizzare**
  - q : **quit**
- **Opzioni**
  - u ***nome_utente***
  - d ***secondi*** : **intervallo di aggiornamento (default 5s)**

Segnali

Come generare un segnale

- Digitare in un terminale delle particolari combinazioni di tasti.
  - `CTRL+C`: interruzione di un processo.
- Eccezioni hardware
  - divisione per zero
  - accesso non valido alla memoria.
  - ...
- I processi possono inviare segnali a se stessi usando la chiamata di sistema `kill()`
- Il kernel può generare segnali per informare i processi quando succede qualcosa di particolare.
  - `SIGPIPE` se un processo tentasse di scrivere su una pipe chiusa dal processo che dovrebbe leggerla.

Elenco dei segnali (1 di 2)

Nome segnale	Numero segnale	Descrizione segnale
SIGHUP	**1**	**Terminali di linea hangup (sospensione)**
SIGINT	**2**	**Interruzione del processo**
SIGQUIT	3	Chiude programma
SIGILL	4	Istruzione illegale
SIGTRAP	5	L'esecuzione del processo ha raggiunto un breakpoint (trap), il debugger puo' informare di questo lo sviluppatore
SIGABRT	6	Interruzione anormale (abort) del processo.
SIGEMT	7	Emulare istruzionieseguite
SIGFPE	8	Eccezione in un numero in virgola mobile
SIGKILL	**9**	**Interruzione immediata. Questo segnale non puo' essere ignorato ed il processo che lo riceve non puo' eseguire delle operazioni di chiusura "morbida".**
SIGBUS	10	Errore di bus: "tentato accesso ad una porzione indefinita di memoria"

Elenco dei segnali (1 di 2)

Nome segnale	Numero segnale	Descrizione segnale
SIGSEGV	11	Errore di segmentazione
SIGSYS	12	Chiamata di sistema errata
SIGPIPE	13	Se un processo che dovrebbe leggere da una pipe termina inaspettatamente, questo segnale viene inviato al programma che dovrebbe scrivere sulla pipe in questione.
SIGALRM	14	Il timer in tempo reale e' scaduto. Segnale sollevato da alarm().
SIGTERM	15	Terminazione del programma; il comando kill invia questo segnale se non diversamente specificato.
SIGURG	16	Sono disponibili dei dati urgenti per il processo su un socket.
SIGSTOP	17	Ferma temporaneamente l'esecuzione del processo: questo segnale non puo' essere ignorato
SIGTSTP	18	Ferma temporaneamente l'esecuzione del processo
SIGCONT	19	Il processo puo' continuare, se era stato fermato.
SIGCHLD	20	Processo figlio terminato o fermato

Elenco dei segnali (2 di 2)

Nome segnale	Numero segnale	Descrizione segnale
SIGTTIN	21	Un processo in esecuzione in background tenta di leggere da terminale
SIGTTOU	22	Un processo in esecuzione in background tenta di scrivere sul terminale
SIGIO	23	I / O su un possibile Descrittore
SIGXCPU	24	Esaurito il tempo di CPU disponibile per il processo
SIGXFSZ	25	Superata la dimensione massima consentita per i file per il processo
SIGVTALRM	26	Un conto alla rovescia impostato per il processo e' terminato. Misura il tempo "virtuale" consumato dal solo processo.
SIGPROF	27	Un conto alla rovescia impostato per il processo e' terminato: misura il tempo di CPU usato dal processo e dal sistema per eseguire azioni istruite dal processo stesso.
SIGWINCH	28	Dimensione della finestra e' cambiato
Siginfo	29	Richiesta di informazioni
SIGUSR1	30	Definito dall'utente
SIGUSR2	31	Definito dall'utente
SIGTHR	32	Thread di interrupt

Segnali prodotti da tastiera

Ctrl+C *(SIGINT)*	interrompe il comando corrente
Ctrl+Z *(SIGSTOP)*	ferma il comando corrente, da continuare con fg in primo piano o in sottofondo con bg
Ctrl+D	esci dalla sessione corrente, simile a exit
Ctrl+W	cancella una parola nella linea corrente
Ctrl+U	cancella l'intera linea
Ctrl+/ *(SIGQUIT)*	uscita invocata da tastiera
Ctrl+R	cicla attraverso la lista dei comandi recenti
!!	ripete l'ultimo comando
exit	esci dalla sessione corrente

kill (1 di 2)

- Segnali inviati da un processo ad un altro processo attraverso il kernel
- Il processo che riceve il segnale fa operazioni di default
- `kill -l` : mostra l'elenco dei segnali disponibili

```
kill –numero/stringa PID
```

kill (2 di 2)

- root può lanciare segnali a tutti i processi
- Gli utenti possono lanciare segnali solo ai processi di cui sono proprietari

killall (1 di 2)

- **Manda un segnale a tutti i processi specificati**
- **SYNOPSIS**

```
killall [-Z,--context pattern]
   [-e,--exact] [-g,--process-group]
   [-i,--interactive] [-q,--quiet]
   [-r,--regexp] [-s,--signalsignal]
   [-u,--user user] [-v,--verbose]
   [-w,--wait] [-I,--ignore-case]
   [-V,--version] [--] name ...
```

killall (2 di 2)

- Segnale di default: `SIGTERM`
- Il segnale puo' essere specificato
  - Per nome: `-HUP` o `–SIGHUP`
  - Per numero: `-1`
- Non uccide se stesso ma può uccidere altri processi di `killall`

`killall`: opzioni principali

- `killall -l`
  - **Lista tutti i segnali**
- `killall -s USR1 proc`
  - **Invia il segnale** `USR1` **al processo** *`proc`*
- `killall -9 proc`
  - **Uccide un processo che non risponde**
  - **(-9 = SIGKILL)**

nohup

- Quando un comando viene impartito in modo che l'esecuzione prosegua in background si pone il problema della sua terminazione o continuazione nel momento in cui si opera un logout.
- Se non si prendono opportune precauzioni, il comando in esecuzione in background viene automaticamente terminato all'atto del logout perchè le shell inviano automaticamente a tutti i processi "discendenti" un segnale di hangup (SIGHUP) e se questo segnale non è stato dichiarato come segnale da ignorare il processo termina.

```
$ nohup comando > outcomando &
```

nohup

- **Se l'utente non provvede a ridirigere esplicitamente l'output di un comando preceduto da** `nohup`**, stdout e stderr vengono automaticamente ridiretti insieme nel file** `nohup.out`**.**
- `nohup` **puo' anche avere l'effetto di incrementare di 5 la priorità del processo**

nohup e shell remota

- **Quando ci si logga con una shell ssh ad un server remoto e si esegue un comando questo è attivo finchè non si effettua un logout dalla macchina.**
- **Utilizzando** nohup **è possibile lasciare attivo il comando eseguito anche quando ci si disconnette dal server remoto amministrato via ssh.**

```
nohup command-name &
```

Shell e Processi

Esecuzione in background

- I comandi di un interprete possono essere eseguiti in parallelo in background
- `&` : esegue un processo in background

```
./loop_inf &
```

- `bg` : manda un processo sospeso in background
- `fg` : manda l'ultimo processo in background in foreground

bg (1 di 2)

- Lanciare `./loop_inf`
- Bloccarlo con SIGTSTP [17](`CTRL+z`)
- Aprire una nuova shell e digitare top
- Digitare `bg` per far ripartire `loop_inf`
- Come argomento del comando `bg` può essere specificato il PID del processo da mandare in background.

bg (2 di 2)

- Lanciare `./loop_inf`
- Bloccarlo con SIGTSTP [17] (`CTRL+z`)
- Lanciare top
- Lanciare il segnale SIGCONT [19] a `loop_inf`

Esercizi (1 di 5)

```
/*loop_inf.c*/
#include <unistd.h>
int main ( void ) {
  while ( 1 ) { }
  return 1;
}
```

Esercizi (2 di 5)

```
/*loop_time.c*/
#include <unistd.h>
#include <time.h>
int main ( void ) {
  while ( 1 ) { time(NULL); }
  return 1;
}
```

Esercizi (3 di 5)

```
/*loop_sleep.c*/
#include <unistd.h>
#include <time.h>
int main ( void ) {
  while ( 1 ) {
     int i;
     sleep (4);
     for ( i = 0; i < 1000000000; i++ );
  }
  return 1;
}
```

Esercizi (4 di 5)

- **Utilizzare il programma `loop_inf.c`**
  - **compilare il programma (`cc -Wall -o <nome_eseguibile> <nome_sorgente_C>`)**
  - **eseguire il programma come utente `osor`**
  - **tramite `top` (*lanciato da utente `root`*) controllare lo stato della CPU**
  - **giustificare l'output di `top`**
  - **modificare il livello di `nice` tramite il comando interattivo di `top`**
  - **come cambia l'output di `top` rispetto a prima (relativamente all'utilizzo della CPU)?**
  - **terminare il programma usando il comando interattivo `kill` da dentro `top`**
- **Utilizzare il programma `loop_time.c`**
  - **compilare ed eseguire come utente `osor`**
  - **controllare le differenze dell'output di `top` (*lanciato da utente `root`*) rispetto a prima**
  - **terminare il programma**

Esercizi (5 di 5)

- Utilizzare il programma `loop_sleep.c`
  - compilare ed eseguire come utente `osor`
  - settare a 1 secondo l'intervallo di update di `top` (*lanciato da utente root*)
  - visualizzare solo i processi che appartengono all'utente con cui il programma è stato lanciato
  - ordinare l'output secondo il campo `time`
  - controllare tramite `top` gli stati in cui passa il processo ed il livello di priorità
  - terminare il processo
- Usare `ps` per ottenere l'elenco tutti i processi in esecuzione sulla macchina,
  - modificando le informazioni in output tramite le opzioni `-o` e `-O`

Soluzioni

- È importante osservare l'utilizzo della CPU. Visto che il programma gira sempre *solo* in spazio utente, la percentuale 'user' va al 100%. Cambiando il nice level, oltre a cambiare il numero nella colonna nice, il consumo di CPU si sposta nel campo nice.
- In questo caso il programma invoca continuamente una syscall, quindi gira anche in modalità kernel. La percentuale di CPU 'system' non raggiunge il 100% perchè comunque il programma in parte gira *anche* in spazio utente (gestione del ciclo).
- In questo caso il processo passa dallo stato run (quando esegue il ciclo interno) allo stato sleep (quando e' bloccato in attesa del timer, per effetto della chiamata sleep()). È interessante notare il livello di priorità: questo cresce quando il processo e' sleeping, mentre decresce quando il processo usa la CPU, cioè durante il ciclo for. Questo perché UNIX decrementa la priorità dei processi che stanno usando molta CPU, e viceversa (politica anti-starvation).