Sistemi operativi

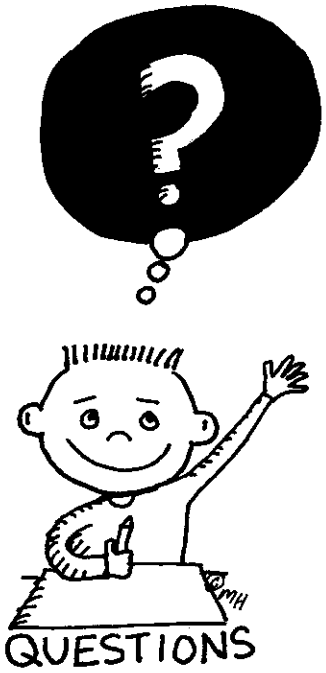
Corso di Laurea Triennale in Ingegneria Informatica

Lezione 5

- Programmazione concorrente
- Processi, fork
- Processi: Gestione, Priorità
- Comandi per gestire processi
- Pianificazione esecuzione

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Domande sulle lezioni passate?



Soluzione esercizi passati

- Da utente (a partire dalla propria home)
  - tar cvzf config.tgz /etc/*conf
  - tar ztvf config.tgz
  - gunzip config.tgz
  - tar xvf config.tar
- Da qualsiasi cartella
  - find /etc/ -name *sys* -size +10c
  - find / -perm -u=s -or -perm -g=s
  - find / -name *tab* -exec cat {} +

Sommario

- Programmazione concorrente
- Processi
  - Operazioni sui processi
    - Creazione, comunicazione, …
    - Gestione dei processi
    - Priorità dei processi (`nice`)
    - Comandi per la gestione dei processi (`ps`, `top`)
- Pianificazione esecuzione

Programmazione concorrente

- Tecniche e strumenti per supportare più attività simultanee in una applicazione software.
- Caratteristica dei sistemi multiprogrammati.

- Consente a più utenti di accedere contemporaneamente ad un sistema informatico
- Consente ad un solo utente l'esecuzione di più programmi simultaneamente
- Consente ad un singolo programma di scomporre la propria attività in più attività concorrenti

Programma vs Processo

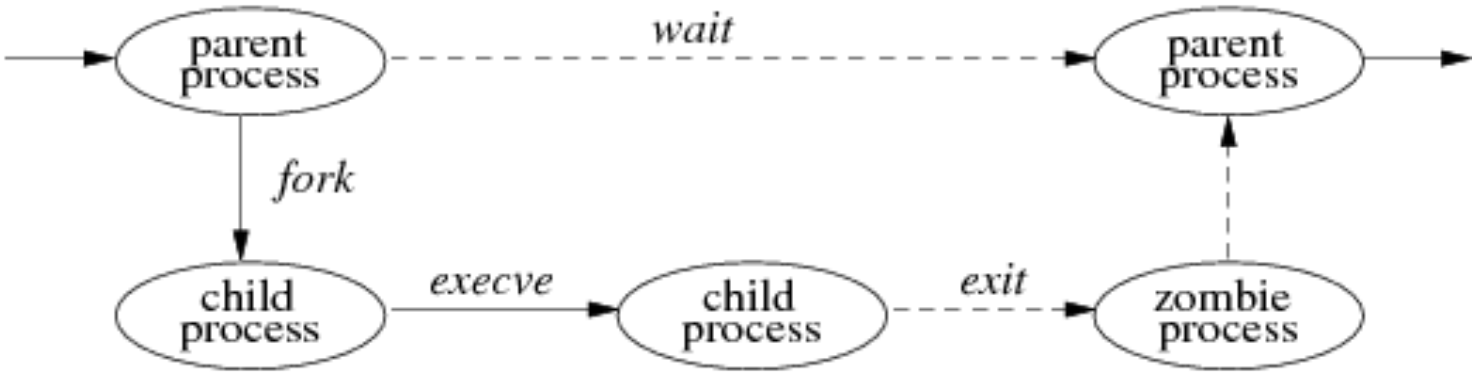
- **Programma** : entità statica che rimane immutata durante l'esecuzione. È costituito dal codice oggetto generato dalla compilazione del codice sorgente.

- **Processo** : entità utilizzata dal sistema operativo per rappresentare una specifica esecuzione di un programma. È un'entità dinamica, che dipende dai dati elaborati e dalle operazioni eseguite.

  Il processo è caratterizzato:

  - dal codice eseguibile
  - dall'insieme di tutte le informazioni che ne definiscono lo stato
    - ⇨ contenuto della memoria indirizzata
    - ⇨ i thread
    - ⇨ i descrittori dei file e delle periferiche in uso.

Processi Unix



Processi

- Processi, sono entità che possono condividere codice (se codice rientrante, in Unix è rientrante).

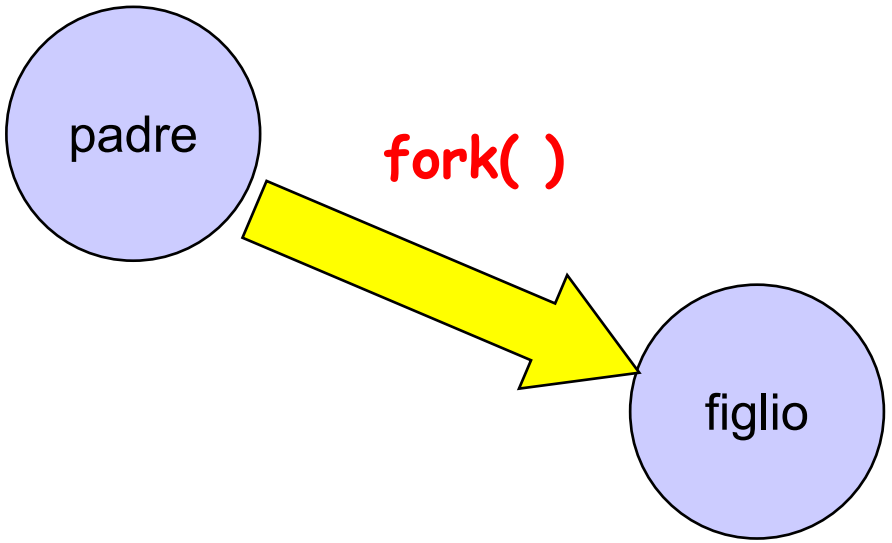
- Il codice è rientrante se può essere condiviso con altri processi, senza che esso venga alterato e se non modifica aree di memoria condivisa o variabili globali

System call per i processi

- Creazione di processi: `fork()`
- Terminazione: `exit()`
- Sospensione in attesa della terminazione di figli: `wait()`
- Sostituzione di codice e dati: `exec..()`

Creazione di processi: fork() - 1

- `fork()` consente a un processo di generare un processo figlio:
  - Padre e figlio condividono lo stesso codice
  - Il figlio *eredita* una copia dei dati (di utente e di kernel) del padre



Creazione di processi: fork() - 2

```
int fork(void);
```

- La fork non richiede parametri
- Restituisce un intero che:
  - Per il processo creato (figlio) vale 0
  - Per il processo padre:
    - ⇨ È un valore > 0 che rappresenta il PID del processo figlio
    - ⇨ È un valore < 0 in caso di errore

Relazione padre-figlio - fork()

- Concorrenza: padre e figlio procedono in parallelo
- Il codice è condiviso
- Lo spazio degli indirizzi è duplicato:
  - ⇨ Ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della fork()
- La user structure è duplicata:
  - ⇨ Le risorse allocate al padre prima della generazione sono condivise col figlio
  - ⇨ Stessa gestione dei segnali per padre e figlio
  - ⇨ Il figlio nasce con lo stesso Program Counter del padre: la prima istruzione eseguita è quella che segue immediatamente la fork()

PPID e PID

`getpid()`

Restituisce il PID del processo

`getppid()`

Restituisce il PPID del processo, cioè il PID del processo padre

Terminazione di processi

- Involontaria (tentativi azioni illegali)
- Volontaria (esempio syscall `exit`)
- Terminazione del processo:
  - Se il processo che termina ha figli in esecuzione, il processo `init` adotta i figli
  - Se il processo termina prima che il padre ne rilevi lo stato di terminazione (con `wait`) il processo passa allo stato *zombie*

Terminazione di processi - **exit()**

```
void exit(int status);
```

- Attraverso il parametro *status* il processo che termina può comunicare al padre informazioni sul suo stato di terminazione
- È sempre una chiamata senza ritorno

- Effetti di una `exit()`:
  - Chiusura dei file aperti
  - Terminazione del processo

Terminazione di processi - **wait()**

```
int wait(int *status);
```

- Lo stato di terminazione può essere rilevato dal processo padre, mediante la syscall `wait()`:
  - Il parametro status è l'**indirizzo** della variabile in cui viene memorizzato lo stato di terminazione del figlio
  - Il risultato della wait è il **pid** del processo terminato, oppure un codice di errore (<0)

Terminazione di processi - **wait()**

- Effetti della syscall `wait(& status)`:
- Il processo che la chiama può avere figli in esecuzione
  - Se tutti i figli non sono ancora terminati, il processo su sospende in attesa della terminazione del primo di essi
  - Se almeno un figlio è già terminato ed il suo stato non è ancora stato rilevato (stato zombie), la `wait` ritorna immediatamente (stato nella variabile `status`)
  - Se non esiste neanche un figlio, la wait non è sospensiva e ritorna un codice di errore (<0)

Sostituzione di codice - **exec..()**

- È possibile sostituire il codice eseguito da un processo mediante una syscall della famiglia `exec()`:

  `execl(), execle(), execclp(),`
  `execv(), execve(), execvp(), ..`

  - Effetto principale di una **`exec..()`**:
    - ⇨ Vengono sostituiti codice e dati del processo che chiama la syscall, con codice e dati di un programma specificato come parametro

Sostituzione di codice - **execl()**

```
int execl(char *pathname, char *arg0,
..., char *argN, (char*) 0);
```

- **`pathname`** è il nome del file contenente il nuovo programma
- **`arg0`** è il nome del programma (`argv[0]`)
- **`arg1,...,argN`** sono gli argomenti da passare al programma
- **`(char*) 0)`** è il puntatore nullo che termina la lista

Effetti dell'esecuzione di **exec**

- Il processo dopo l'`exec()`:
  - Mantiene la stessa process structure (salvo le informazioni relative al codice)
    - ⇨ Stesso pid, stesso ppid (pid padre)
  - Ha codice, dati globali, stack e heap nuovi
  - Riferisce una nuova text structure
  - Mantiene user area (a parte PC e informazioni legate al codice) e stack del kernel:
    - ⇨ Mantiene le stesse risorse
    - ⇨ Mantiene lo stesso environment (salvo che si usi **`execle()`, `execve()`**)

Esempio di fork() ed execl()

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char * argv[]) {
  int pid, status;
  pid=fork();
  if (pid==0) {
    execl("/bin/ls", "ls", "-l", (char *) 0);
    printf("exec fallita!\n");
    exit(1);
  }
  else if (pid>0) {
    pid=wait(&status);

    /* Gestione dello stato */

    }
  else printf("fork fallita!\n");
}
```

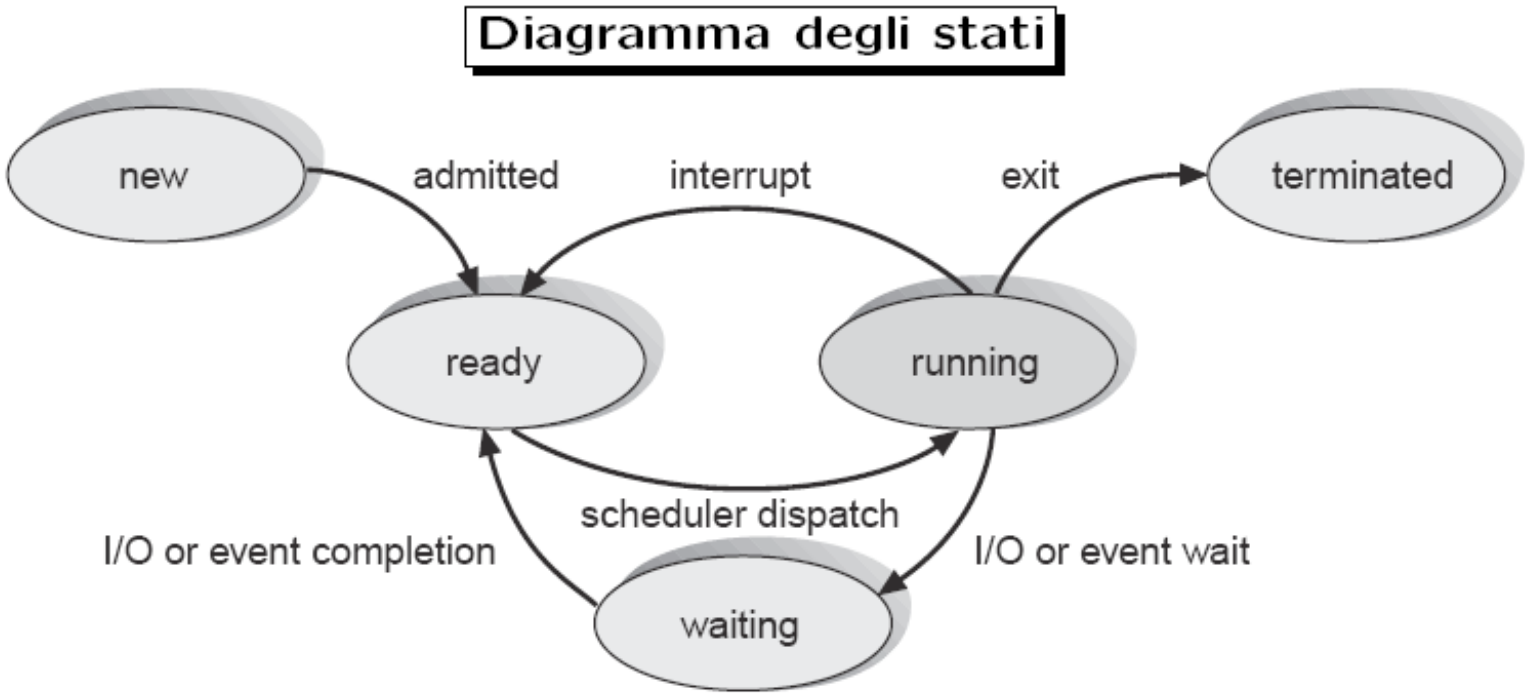
Codice nella cartella esempi

- fork1.c
  - Called once, returns twice
- fork2.c
  - Uso di getpid() e getppid()
- exec1.c
  - Uso di execv (path ...)
- fdtest1.c
  - Read e Write dagli stessi file

Codice nella cartella esempi

- die1.c
  - Padre termina prima del figlio (./die)
- die2.c
  - Figlio termina prima del padre (./die2 &)
- die3.c
  - Signal handler in azione (./die3 &)

Gestione dei processi



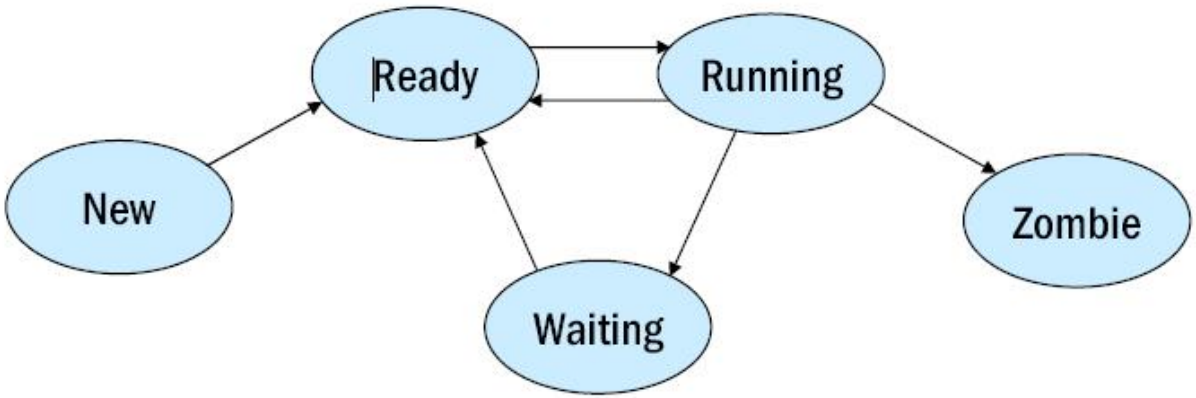
Processo

- Processo : programma singolo nel momento in cui viene eseguito

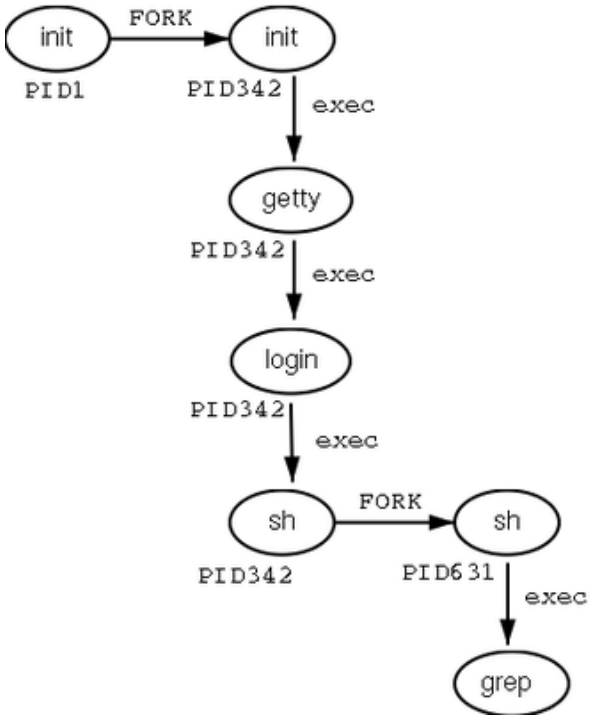
- I processi di dividono in:
  - Utente
  - Sistema

Stato di un processo

- Creazione di un processo
  - chiamata di sistema `fork()`
  - soltanto da un processo già attivo
- Terminazione di un processo
  - chiamata di sistema `exit()`
  - “zombie”: processo che ha completato la sua esecuzione ma che compare ancora nella tabella dei processi (exit status)



init



- `init (/sbin/init)` e' la radice dell'albero dei processi
  - creato dal kernel al termine del bootstrap
  - PID = 1
  - Non ha padre
- Se un processo termina i suoi figli diventano figli di `init`

Identificazione dei processi

- PID: identificatore univoco di processo
- PGID: identificatore di gruppo-processi
- UID: identificatore di utente
- GID: identificatore di gruppo-utenti
- PPID: Parent PID

PID

- Numero a 16 bit (di tipo `pid_t`) assegnato sequenzialmente dal kernel per ogni nuovo processo creato
- >=0
- Univoco
- Riciclato quando termina
- =0 scheduler
- =1 init
- =2 pagedaemon

ID e privilegi

- A ogni processo sono associati degli identificatori di utente (*user-IDs*) e di gruppo (*group-IDs*) che determinano i suoi privilegi → **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
  - I valori *real user-id* e *real group-id* non cambiano per tutta la sessione di login
  - solo root ha il potere di cambiarli

Effective ID

- EUID, EGID e supplementary GID: UID e GID considerati per determinare i privilegi per l'accesso ai file
  - Possono non coincidere con RUID e RGID se 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 l'EUID (EGID) diventa quello dell'*owner* (*group owner*) del file, quando viene invocato con il comando **exec.**

`setuid` e `setgid`

- `int setuid(uid_t uid);`
- `int setgid(gid_t gid);`

- Cambiano real user/group ID ed effective user/group ID

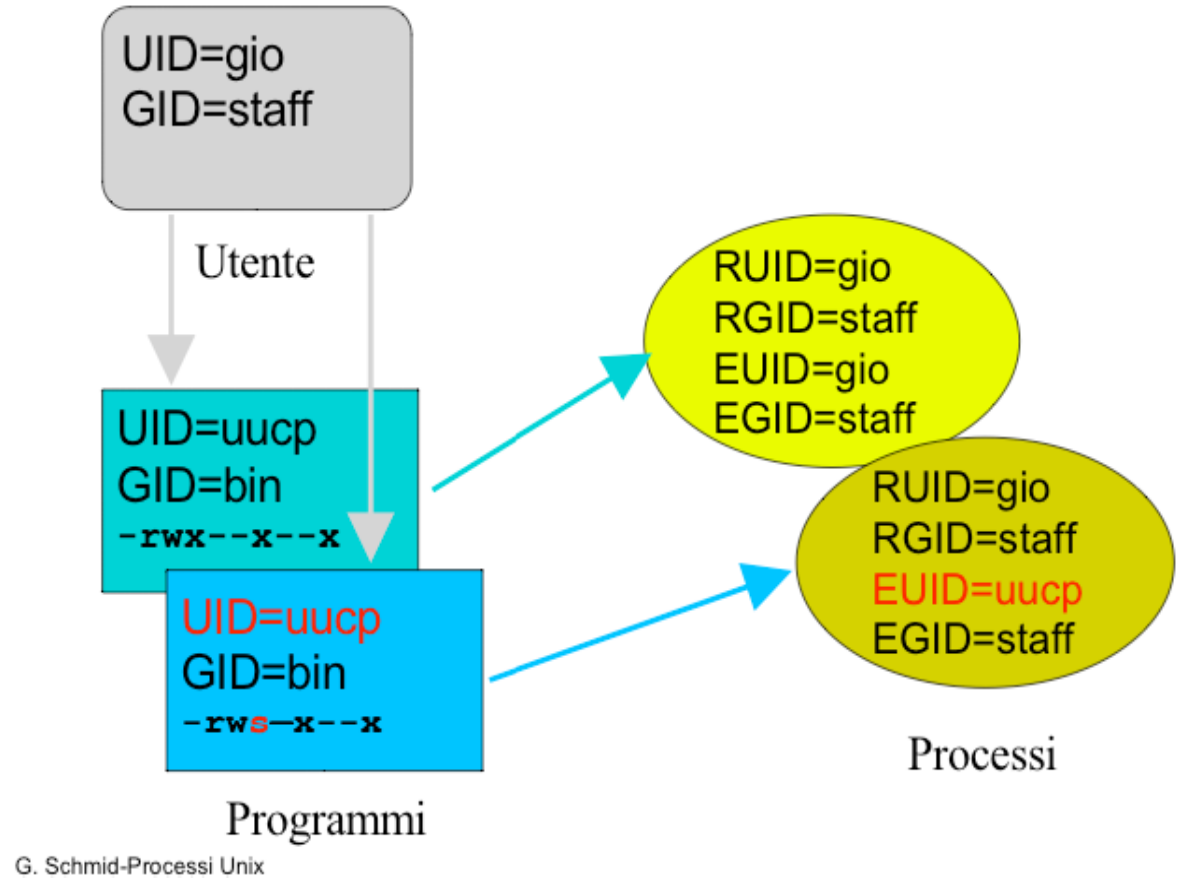
- Idem per `setgid`

- NB: se invocato da un processo con ID del super utente, una volta cambiato EUID il processo non può riacquisire i privilegi di root!

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
- Esempio → 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
```



G. Schmid-Processi Unix

Get

- `pid_t getpid(void)` : PID del processo
- `pid_t getppid(void)` : PPID del processo
- `uid_t getuid(void)` : RUID
- `uid_t geteuid(void)` : EUID
- `gid_t getgid(void)` : RGID
- `gid_t getegid(void)` : EGID
- Non possono fallire

Esempio

```
#include <unistd.h>
int main(){
      uid_t uid, euid;
      gid_t gid, egid;
      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);
}
```

Priorità dei Processi

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Priorità dei processi

- Lo *scheduler* UNIX organizza l'esecuzione dei processi in base al livello di priorità.
- La priorità viene calcolata in base a:
  - tipo di processo
  - comportamento del processo
  - variabile NICE.

```
ps -l
```

Priorità

- Priorità:
  - 0 – 99 – Priorità Real Time
  - 100 – 139 – Priorità per i processi dell'utente
  - -20 e 19 – valori assunti da `nice`
  - **Più alta è la priorità → più "lentamente" viene eseguito il processo.**
- `nice` viene sommata nel calcolo della priorità.
- L'utente (normale o root) non può intervenire sulla priorità ma solo sul NICE

nice

- Quando viene creato un nuovo processo, alla variabile NICE viene assegnato il valore di nice del padre (di solito 0)
- L'utente può richiedere una variazione con il comando nice.
- L'utente (non root) può solo aumentare il NICE value.
- Non è 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

Attenzione ai valori negativi!

- PR = 20 + NI

PR si mappa nell'intervallo di priorità 100-139. A volte si può trovare come PR = 120 + NI

Effetti di `nice`

- Se la variabile NICE aumenta il suo valore allora il processo esegue più lentamente alleggerendo il sistema.

- Un uso appropriato è 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.
- `renice` accetta opzioni per specificare i processi non solo per PID ma anche per utente o gruppo di processi.

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

Esempio

Lanciare il comando `bzip2` in background

```
$ bzip2 fileTantoGrande &
```

Guardare il valore di NICE

```
$ ps -xl
```

Cambiare il livello di NICE

```
$ renice valore PID
```

Verificare il cambiamento

Comandi per la gestione dei Processi

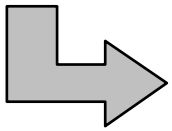
Fabio Del Vigna - fabio.delvigna@for.unipi.it

`ps` (1 di 3)

- **`ps`** : visualizza i 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 : swappato
- N : NICE > 0

`ps` (2 di 3) opzioni

- `x` : visualizza anche i processi che non provengono da terminali
- `u` : indica l'utente a cui appartiene ogni processo
- `a` : visualizza i processi di tutti gli utenti
- `O` : mostra i campi elencati di seguito, più quelli di default
- `o` : precisa le *sole* colonne da mostrare
- `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
daniel 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 totale in cui ha usato la CPU
- COMMAND : percorso che ha creato il processo

`top` (1 di 3)

- `top` : visione dinamica dei processi
  - load average : utilizzo della CPU nell'ultimo minuto, ultimi 5 e ultimi 15
  - CPU state
    - ⇨ user : usata dagli utenti
    - ⇨ system : usata dalle system call
    - ⇨ interrupt : carico dato dalle interruzioni
    - ⇨ nice : cresce quando cambio NICE
    - ⇨ idle : usata dalla dummy()
  - Mem
    - ⇨ Total
    - ⇨ Used
    - ⇨ Shared
    - ⇨ Free

top (2 di 3)

- PID
- USERNAME
- THR : thread
- PRI : priorità
- NICE
- SIZE : dimensione in memoria virtuale
- RES : resident
- STATE
- TIME
- WCPU : media veloce
- 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)

Pianificazione dei processi

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Pianificazione dei processi

- Esecuzione in date e orari stabiliti
- Demone `cron` → controlla esecuzioni pianificate
- `crontab`: configurazione cron
- Solitamente si ha:
  - Un file per ogni utente
  - Uno generale per tutto il sistema

cron

- **Demone** (in background)
- File `crontab` collocati in
  - **`/etc/crontab`**
  - **`/var/cron/tabs/`*`nome_utente`***

Comando `crontab`

`crontab [opzioni]`

- creare o modificare file `crontab`
- File `crontab` usati da `cron` per eseguire i comandi indicati

Solo root può agire sul file `crontab` di un altro utente

Opzioni al comando `crontab`

`[-u utente] file`

- Sostituisce il file `crontab`

`-l`

- Visualizza il file `crontab`

`-e`

- Crea o modifica il file `crontab`

`-r`

- Cancella il file `crontab`

Variabili di ambiente

- `SHELL`
  - Quale shell esegue i comandi (es. `/bin/sh`)

- `MAILTO`
  - Destinatario dei messaggi
  - "" non viene inviato nessun messaggio

Formato del **file** `crontab`

```
23 0-23/2 * * * echo "Ciao"
```

- campi **separati** da spaziature
- istante di esecuzione
  - **`minuto, ora, giorno, mese, giorno della settimana`**
- `utente`
  - solo per `/etc/crontab`
- `comando`
  - senza redirezione l'output viene inviato per e-mail

Caratteri speciali

- *
  - qualsiasi valore
- –
  - delimitare valori (es. 1-3)
- ,
  - separare singoli valori (es. 2,5)
- /
  - esprimere una granularità (es. /8). Indica uno step temporale da seguire
- Es: */2 sulle ore esegue il comando ogni due ore

Esempio /var/cron/tabs/*

```
# Utilizza «/bin/sh» per eseguire i comandi, indipendentemente da
# quanto specificato all'interno di «/etc/passwd».
SHELL=/bin/sh
# Invia i messaggi di posta elettronica all'utente «fede»,
# indipendentemente dal proprietario di questo file crontab.
MAILTO=fede
# Esegue 5 minuti dopo la mezzanotte di ogni giorno.
5 0 * * * $HOME/bin/salvataggiodati
# Esegue alle ore 14:15 del primo giorno di ogni mese.
# L'output viene inviato tramite posta elettronica all'utente
«tizio».
15 14 1 * * $HOME/bin/mensile
# Esegue alle 22 di ogni giorno lavorativo (da lunedì al venerdì).
# In particolare viene inviato un messaggio di posta elettronica a
«fede».
0 22 * * 1-5 mail -s "Sono le 22" fede
# Esegue 23 minuti dopo mezzanotte, dopo le due, dopo le
quattro,...,
# ogni giorno.
23 0-23/2 * * * echo "Ciao ciao"
# Esegue alle ore 04:05 di ogni domenica.
5 4 * * 0 echo "Buona domenica"
```

Esempio /etc/crontab

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
# Run any at jobs every minute
* * * * * root [ -x /usr/sbin/atrun ] &&
/usr/sbin/atrun
# run-parts è un programma che avvia tutti
gli eseguibili contenuti nella directory
indicata come argomento
01 * * * * root run-parts /etc/cron.hourly
02 1 * * * root run-parts /etc/cron.daily
02 2 * * 0 root run-parts /etc/cron.weekly
02 3 1 * * root run-parts /etc/cron.monthly

# esegue uno script di backup ogni 10 minuti
*/10 * * * * root /root/bin/backupscript
```

Note

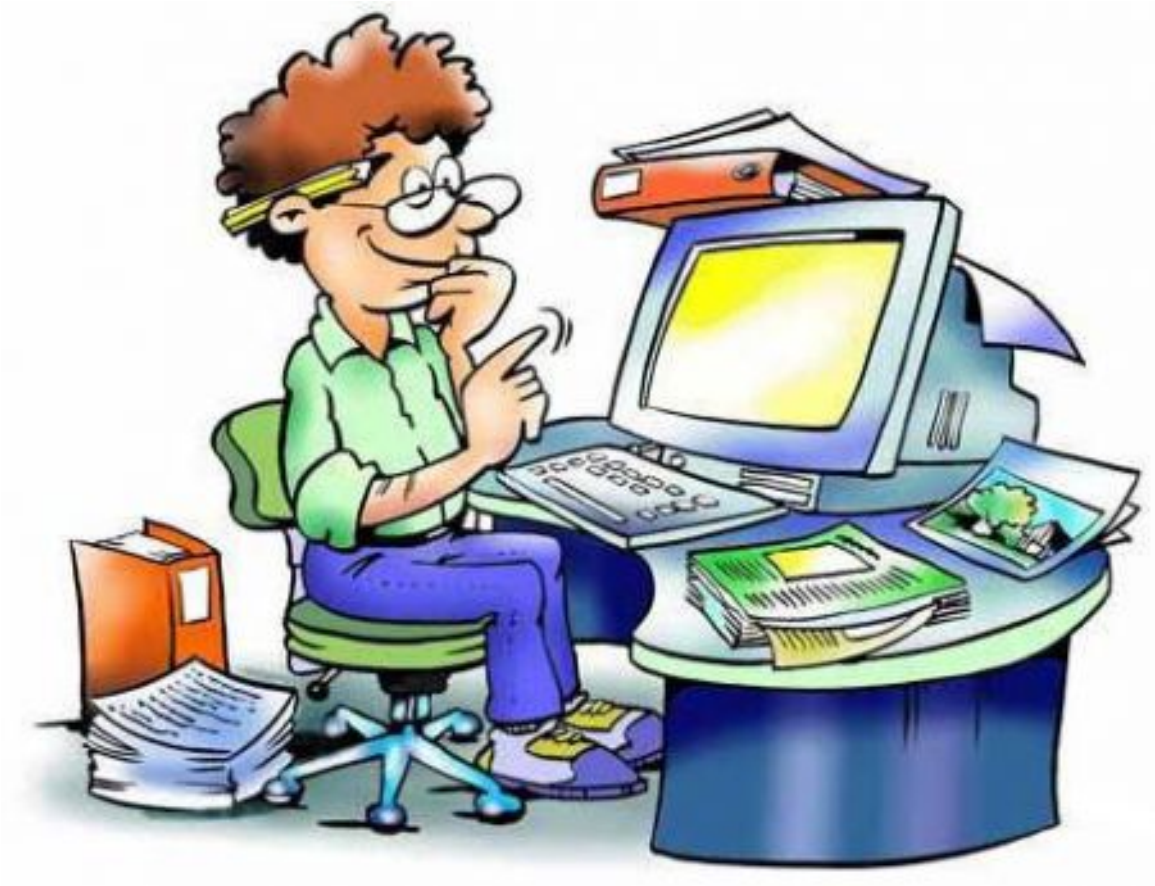
`run-parts /etc/periodic/hourly`

- avvia tutto quello che è nella directory

Per inserire un'elaborazione nei momenti più comuni, basta metterla nella directory che rappresenta la cadenza desiderata.

- Alcune versioni richiedono un newline prima di EOF per eseguire l'ultimo comando

Esercizi



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 `utentenuovo`
  - 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 `sisoper`
  - 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 `sisoper`
  - 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`