Sistemi operativi

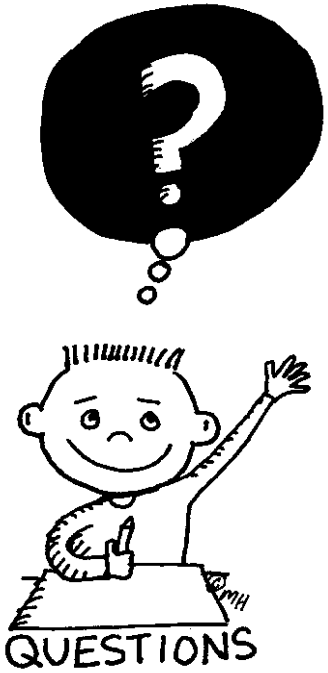
Corso di Laurea Triennale in Ingegneria Informatica

Lezione 8

- `Semaphore.h`
- `Segnali`
- `Shell e processi`

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

Domande sulle lezioni passate?



Sommario

- Semafori generali
  - <semaphore.h>
- Segnali
  - descrizione dei segnali
  - invio di segnali (`kill`, `killall`)
- Shell e processi (`fg`, `bg`)
- Invio segnali da tastiera (^C, ^D)
- Segnali (gestione)

Semafori classici

Semafori

- I semafori sono primitive fornite dal sistema operativo per permettere la sincronizzazione tra processi e/o thread.
- Implementati attraverso dei contatori.
- Per queste primitive è garantita l'atomicità.
- Ogni modifica o check del valore di un semaforo può essere effettuata senza sollevare "*race conditions*".

Race condition

- Più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

  - Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
  - Estremamente pericolose: portano al malfunzionamento dei processi cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
  - difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

Mutex vs Semaforo (1 di 2)

- Il mutex è un tipo definito "ad hoc" per gestire la mutua esclusione quindi il valore iniziale può essergli assegnato anche in modo statico mediante la macro `PTHREAD_MUTEX_INITIALIZER`.
- Al contrario un semaforo come il `sem_t` deve essere di volta in volta inizializzato dal programmatore col valore desiderato.

Mutex vs Semaforo (2 di 2)

- Un semaforo può essere impiegato come un mutex
- Differenza sostanziale: un mutex deve sempre essere sbloccato dal thread che lo ha bloccato, mentre per un semaforo l'operazione `post` può **non** essere eseguita dal thread che ha eseguito la chiamata `wait`.

inizializzo un mutex;

```
pthread_mutex_lock(&mutex);
```

sezione critica

```
pthread_mutex_unlock(&mutex);
```

inizializzo un semaforo (1);

```
sem_wait(&sem);
```

sezione critica

```
sem_post(&sem);
```

Semafori classici (generali) (1 di 2)

- Semafori il cui valore può essere impostato dal programmatore
  - utilizzati per casi più generali di sincronizzazione
  - esempio: produttore consumatore
- Interfaccia
  - operazione `wait`
  - operazione `post`

Semafori classici (generali) (2 di 2)

- Semafori classici e standard POSIX
  - non presenti nella prima versione dello standard
  - introdotti insieme come estensione real-time con lo standard IEEE POSIX 1003.1b (1993)
- Utilizzo
  - associati al tipo `sem_t`
  - includere l'header

```
#include <semaphore.h>
#include <errno.h>
```

Creazione semaforo

- `sem_t`: tipo di dato associato al semaforo

```
sem_t sem;
```

Inizializzazione (1 di 2)

```
int sem_init(      sem_t *sem, int pshared,
                   unsigned int value )
```

- I semafori richiedono un'inizializzazione esplicita da parte del programmatore
- `sem_init` serve per inizializzare il valore del contatore del semaforo specificato come primo parametro

Inizializzazione (2 di 2)

- `sem_t *sem`
  - puntatore al semaforo da inizializzare, cioè l'indirizzo dell'oggetto semaforo sul quale operare
- `int pshared`
  - flag che specifica se il semaforo è condiviso fra più processi
  - se 1 il semaforo è condiviso tra processi
  - se 0 il semaforo è privato del processo e può essere condiviso solo da thread
- `unsigned int *value`
  - valore iniziale da assegnare al semaforo
- Valore di ritorno
  - `0` in caso di successo,
  - `-1` altrimenti con la variabile `errno` settata a `EINVAL` se il semaforo supera il valore `SEM_VALUE_MAX`

Interfaccia `wait` (1 di 2)

- Consideriamo il semaforo come un intero, sul cui valore la funzione `wait` esegue un test
- Se il valore del semaforo è minore o uguale a zero (*semaforo rosso*), **la** `wait` **si blocca**, forzando un cambio di contesto a favore di un altro dei processi pronti che vivono nel sistema
- Se il test ha successo cioè se il semaforo presenta un valore maggiore od uguale ad `1` (*semaforo verde*), la `wait` decrementa tale valore e ritorna al chiamante, che può quindi procedere nella sua elaborazione.

```
void wait (semaforo s) {
   s.count--;
   if (s.count < 0)
        <cambio di contesto>;
}
```

Interfaccia `wait` (2 di 2)

- Due varianti
  - `wait`: bloccante (standard)
  - `trywait`: non bloccante (utile per evitare deadlock)

wait

```
int sem_wait( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da decrementare
- Valore di ritorno
  - `0` in caso di successo
  - `-1` in caso di fallimento e `errno` viene settato

trywait

```
int sem_trywait( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da decrementare
- Valore di ritorno
  - `0` in caso di successo
  - `-1` se il semaforo ha valore `0`
    - ⇨ setta la variabile `errno` a `EAGAIN`

Interfaccia `post`

- L'operazione di `post` incrementa il contatore del semaforo
- Se a seguito di tale azione il contatore risultasse ancora minore od uguale a zero, significherebbe che altri processi (thread) hanno iniziato la `wait` ma hanno trovato il semaforo rosso
- Se ci sono pocessi (thread) bloccati nella **coda del semaforo**, la `post` sveglia quindi uno di questi;

```
void post(semaforo s) {
   s.count++;
   if (s.count <= 0)
        <sveglia processo>;
}
```

sem_post

```
int sem_post( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da incrementare
- Valore di ritorno
  - `0` in caso di successo
  - `-1` altrimenti con la variabile `errno` settata in base al tipo di errore
    - ⇨ `sem_post` restituisce `EINVAL` se il semaforo supera il valore `SEM_VALUE_MAX` dopo l'incremento

sem_destroy

```
int sem_destroy( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da distruggere
- Valore di ritorno
  - `0` in caso di successo
  - `-1` altrimenti con la variabile `errno` settata in base al tipo di errore
    - ⇨ `sem_destroy` restituisce `EBUSY` se almeno un thread è bloccato sul semaforo

sem_getvalue

- **Serve per poter leggere il valore attuale del contatore del semaforo**

```
int sem_getvalue( sem_t *sem, int *sval )
```

- `sem_t *sem`
  - puntatore del semaforo di cui leggere il valore
- `int *sval`
  - valore del semaforo
- Valore di ritorno
  - sempre `0`

Esempio 1: lettori e scrittori (1 di 5)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define LUN 20
#define CICLI 1
#define DELAY 100000
struct {
  char scritta[LUN+1];
  /* Variabili per la gestione del buffer */
  int primo, ultimo;
  /* Variabili semaforiche */
  sem_t mutex, piene, vuote;
} shared;
void *scrittore1(void *);
void *scrittore2(void *);
void *lettore(void *);
```

Esempio 1: lettori e scrittori (2 di 5)

```
int main(void) {
  pthread_t s1TID, s2TID, lTID;
  int res, i;
  shared.primo = shared.ultimo = 0;
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.piene, 0, 0);
  sem_init(&shared.vuote, 0, LUN);
  pthread_create(&lTID, NULL, lettore, NULL);
  pthread_create(&s1TID, NULL, scrittore1, NULL);
  pthread_create(&s2TID, NULL, scrittore2, NULL);
  pthread_join(s1TID, NULL);
  pthread_join(s2TID, NULL);
  pthread_join(lTID, NULL);

 printf("E' finito l'esperimento ....\n");
}
```

Esempio 1: lettori e scrittori (3 di 5)

```
void *scrittore1(void *in) {
  int i, j, k;
  for (i=0; i<CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.vuote); /* Controllo che il buffer non sia pieno */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      shared.scritta[shared.ultimo] = '-';       /* Operazioni sui dati */
      shared.ultimo = (shared.ultimo+1)%(LUN);
      sem_post(&shared.mutex);                        /* Libero il mutex */
      sem_post(&shared.piene);    /* Segnalo l'aggiunta di un carattere */
      for(j=0; j<DELAY; j++);              /* ... perdo un po' di tempo */
    }
  }
  return NULL;
}
```

Esempio 1: lettori e scrittori (4 di 5)

```
void *scrittore2(void *in) {
  int i, j, k;
  for (i=0; i<CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.vuote); /* Controllo che il buffer non sia pieno */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      shared.scritta[shared.ultimo] = '+';       /* Operazioni sui dati */
      shared.ultimo = (shared.ultimo+1)%(LUN);
      sem_post(&shared.mutex);                    /* Libero il mutex */
      sem_post(&shared.piene);    /* Segnalo l'aggiunta di un carattere */
      for(j=0; j<DELAY; j++);             /* ... perdo un po' di tempo */
    }
  }
  return NULL;
}
```

Esempio 1: lettori e scrittori (5 di 5)

```
void *lettore(void *in) {
  int i, j, k; char local[LUN+1]; local[LUN] = 0;      /* Buffer locale */
  for (i=0; i<2*CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.piene); /* Controllo che il buffer non sia vuoto */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      local[k] = shared.scritta[shared.primo];   /* Operazioni sui dati */
      shared.primo = (shared.primo+1)%(LUN);
      sem_post(&shared.mutex);                        /* Libero il mutex */
      sem_post(&shared.vuote);     /* Segnalo che ho letto un carattere */
      for(j=0; j<DELAY; j++);                /* ... perdo un pò di tempo */
    }
        printf("Stringa = %s \n", local);
  }
  return NULL;
}
```

Curiosità: ^M nei file di testo

- ^M nei file di testo (a fine riga) nasce da un problema nella gestione dell'accapo tra diversi sistemi
- Per rimuoverlo si può usare il programma **tr**

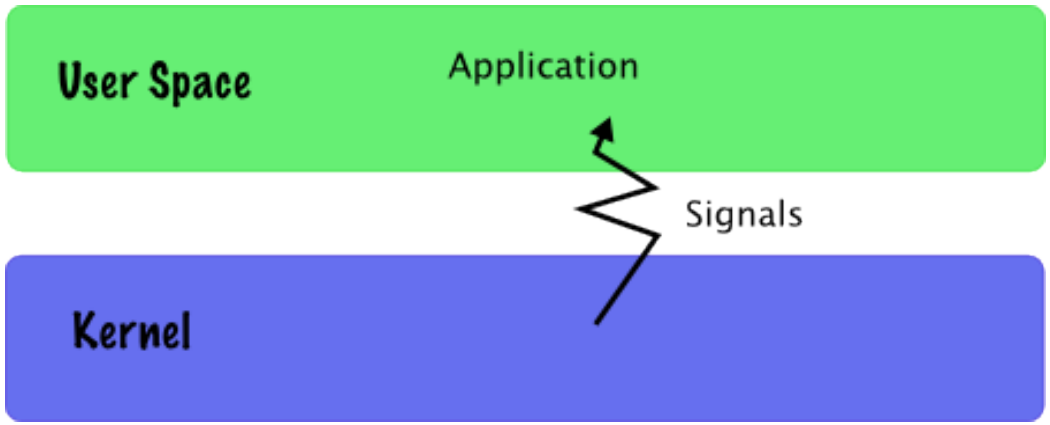
```
cat file.c | tr -d "\r" > file_no_carriage.c
```

```
tr -d "\r" < file.c > file_no_carriage.c
```

- oppure

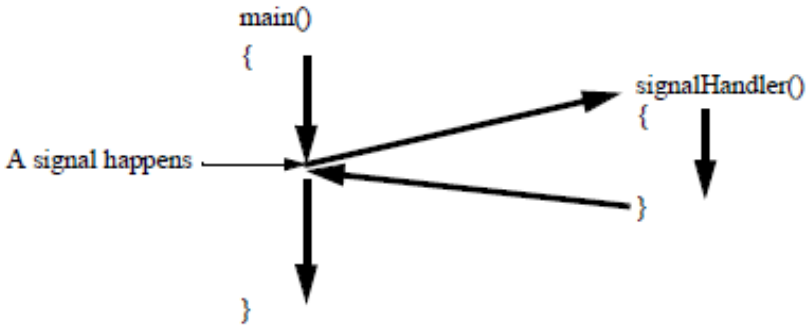
```
dos2unix file.c > file_nc.c
```

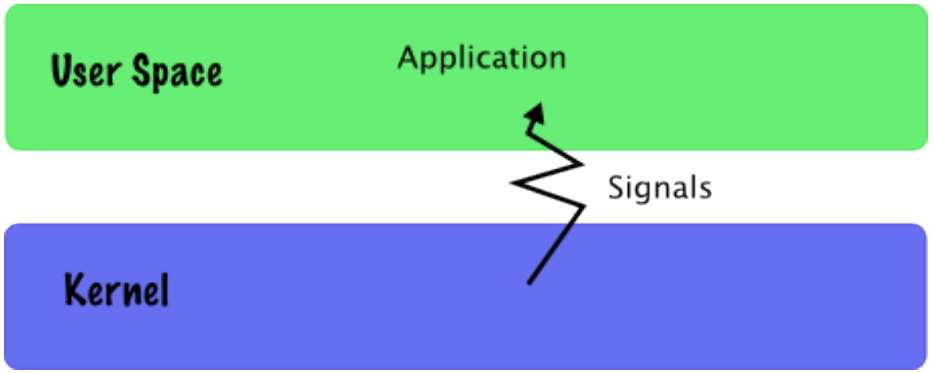
Segnali

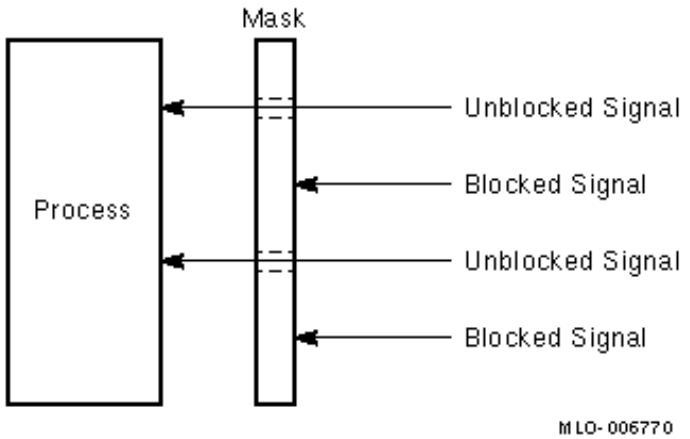


Segnali

- Un segnale è un interrupt software
- Si possono bloccare i segnali (maschera)







Come generare un segnale

- Digitare delle 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 o altri processi usando la chiamata di sistema `kill()`
- Il kernel genera segnali per informare i processi che succede qualcosa
  - `SIGPIPE` se un processo tenta di scrivere su una pipe chiusa dal processo che dovrebbe leggerla.

Elenco dei segnali (1 di 3)

Nome segnale	# 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 può informare di questo lo sviluppatore
SIGABRT	6	Interruzione anormale (abort) del processo.
SIGEMT	7	Emulare istruzioni eseguite
SIGFPE	8	Eccezione in un numero in virgola mobile
SIGKILL	9	Interruzione immediata. Questo segnale non può essere ignorato ed il processo che lo riceve non può eseguire delle operazioni di chiusura "morbida".
SIGBUS	10	Errore bus: "accesso a una porzione indefinita di memoria"

Elenco dei segnali (2 di 3)

Nome segnale	# segnale	Descrizione segnale
SIGSEGV	11	Errore di segmentazione
SIGSYS	12	Chiamata di sistema errata
SIGPIPE	13	Se un processo che legge 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 è scaduto. Sollevato da alarm().
SIGTERM	15	Terminazione del programma; il comando kill invia questo segnale se non diversamente specificato.
SIGURG	16	Disponibili dei dati urgenti per il processo su un socket.
SIGSTOP	17	Ferma temporaneamente l'esecuzione del processo: questo segnale non può essere ignorato
SIGTSTP	18	Ferma temporaneamente l'esecuzione del processo
SIGCONT	19	Il processo può continuare, se era stato fermato.
SIGCHLD	20	Processo figlio terminato o fermato

Elenco dei segnali (3 di 3)

Nome segnale	# segnale	Descrizione segnale
SIGTTIN	21	Un processo in background tenta di leggere da terminale
SIGTTOU	22	Un processo 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 consentita per i file per il processo
SIGVTALRM	26	Un conto alla rovescia impostato per il processo è terminato. Misura il tempo "virtuale" usato dal solo processo.
SIGPROF	27	Un conto alla rovescia impostato per il processo è terminato: misura il tempo di CPU usato dal processo e dal sistema per eseguire azioni istruite dal processo stesso.
SIGWINCH	28	Dimensione della finestra è 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

```
killall [...] [-s,--signalsignal] [...]
  [-u,--user user] name ...
```

`killall` (2 di 2)

- Segnale di default: `SIGTERM`
- Il segnale può 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)

Segnali <*signal.h*>

- I segnali sono una sorta di interruzione software inviata ai processi
- I segnali possono essere legati a diverse cause
  - Kernel
  - Altri processi
  - Interrupt espliciti dell'utente

Inviare segnali

- Command line
  - **`kill -SIGNAME pid`**
  - es. `kill -SIGSTOP 698`

- System call
  - **`int kill (pid_t pid, ing sig)`**
  - es. `kill(my_pid, SIGSTOP);`

Azioni di default

- Ogni segnale ha un'azione di default, che può essere (dopo la ricezione):
  - Segnale scartato
  - Terminazione processo
  - Scrittura di un "*core file*", poi terminazione processo (es. SIGABRT)
  - STOP del processo

Gestione segnali

- Un processo può:
  - Lasciare l'azione di default
  - Bloccare il segnale (per i segnali bloccabili)
  - Intercettare il segnale con un handler

- Per intercettare il segnale bisogna “settare” un handler per quel segnale

Settare un handler `signal( )`

- **`signal (int sig, void (*func()))`**

- func() può essere:
  - SIG_DFL – azione di default, termina il processo
  - SIG_IGN – ignora il segnale (salvo SIGKILL)
  - L'indirizzo di una funzione (specificata dall'utente)
  - esempio: `signal(SIGINT, SIG_IGN);`

Catch ctrl-c

File: catch-ctrl-c.c

```
#include <signal.h>

void catch_int(int sig_num) {
        signal(SIGINT, catch_int); /* re-set the signal
handler */
        printf("Don't do that");  /* and print the message
*/
        fflush(stdout);
}

...

/* set the INT (Ctrl-C) signal handler to 'catch_int' */
signal(SIGINT, catch_int);
for ( ;; )

    pause();
```

Mascherare segnali `sigprocmask()`

```
int sigprocmask(int how, const sigset_t *set,
                sigset_t *oldset);
```

- Ogni processo in Unix ha la propria maschera dei segnali
- La maschera di ogni processo è memorizzata nello spazio del kernel
- **`set`** è una maschera preparata

Agire sulla maschera `set`

- Per agire sul set dei segnali si usano funzioni della famiglia
- `Sigsetops():`
  - `sigaddset()`
  - `sigdelset()`
  - `sigismember()`
  - `sigfillset()`
  - `sigemptyset()`
- Esempio d'uso → file: `count-ctrl-c.c`

Attenzione!

- Per come abbiamo lavorato sui segnali è possibile che emergano situazioni di “race”.
- Per evitarle usare la funzione sigaction() invece che signal()
- Consultate la corrispondente pagina di manuale e esercitatevi (a casa) sul suo uso

Timer e segnali

- Chiamata di sistema **`alarm()`**
- Permette di richiedere l'invio da parte del sistema di un segnale ***ALRM*** al processo che la invoca

```
unsigned int alarm(unsigned int seconds)
```

Vedere anche `setitimer()` che associa 3 timer a ogni processo

file: `use-alarms.c`

NOTE ai segnali

- Handler corti
- Mascherare correttamente
- Attenzione coi segnali di "*fault*" (`SIGBUS`, `SIGSEGV`, `SIGFPE`)
- Attenzione coi timer
- I segnali non sono pensati per realizzare ambienti "*event driven*"

Shell e Processi

(esecuzione in Background ...)

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

Esecuzione in background

- I comandi di un interprete possono essere eseguiti 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*

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.
- 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).
- Se questo segnale è stato dichiarato come segnale da ignorare il processo non termina.

$ nohup *comando* > *outcomando* &

`nohup` e redirezione

- 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`.

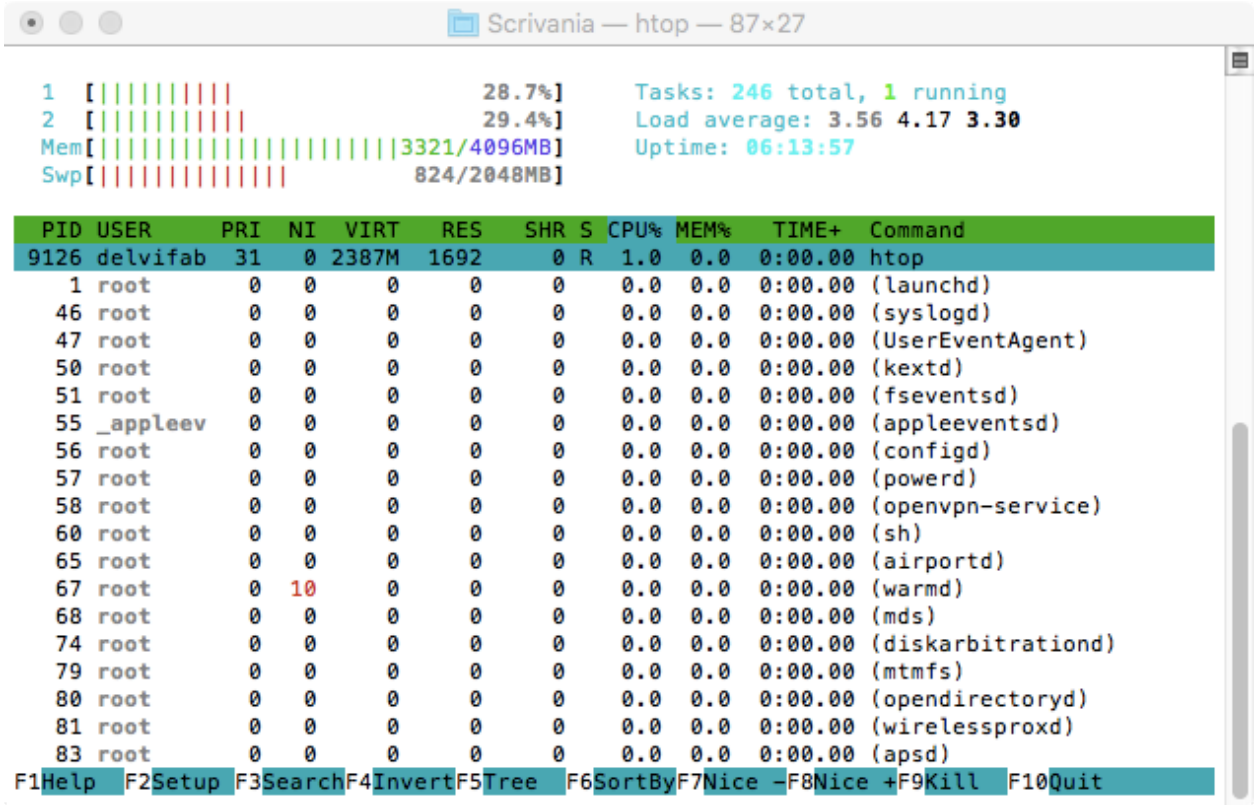
`nohup` e shell remota

- Se si esegue un comando su una macchina remota tramite *ssh,* il comando resta attivo finché non si effettua un logout dalla macchina.
- Utilizzando `nohup` è possibile lasciare attivo il comando eseguito anche quando ci si disconnette dalla macchina remota

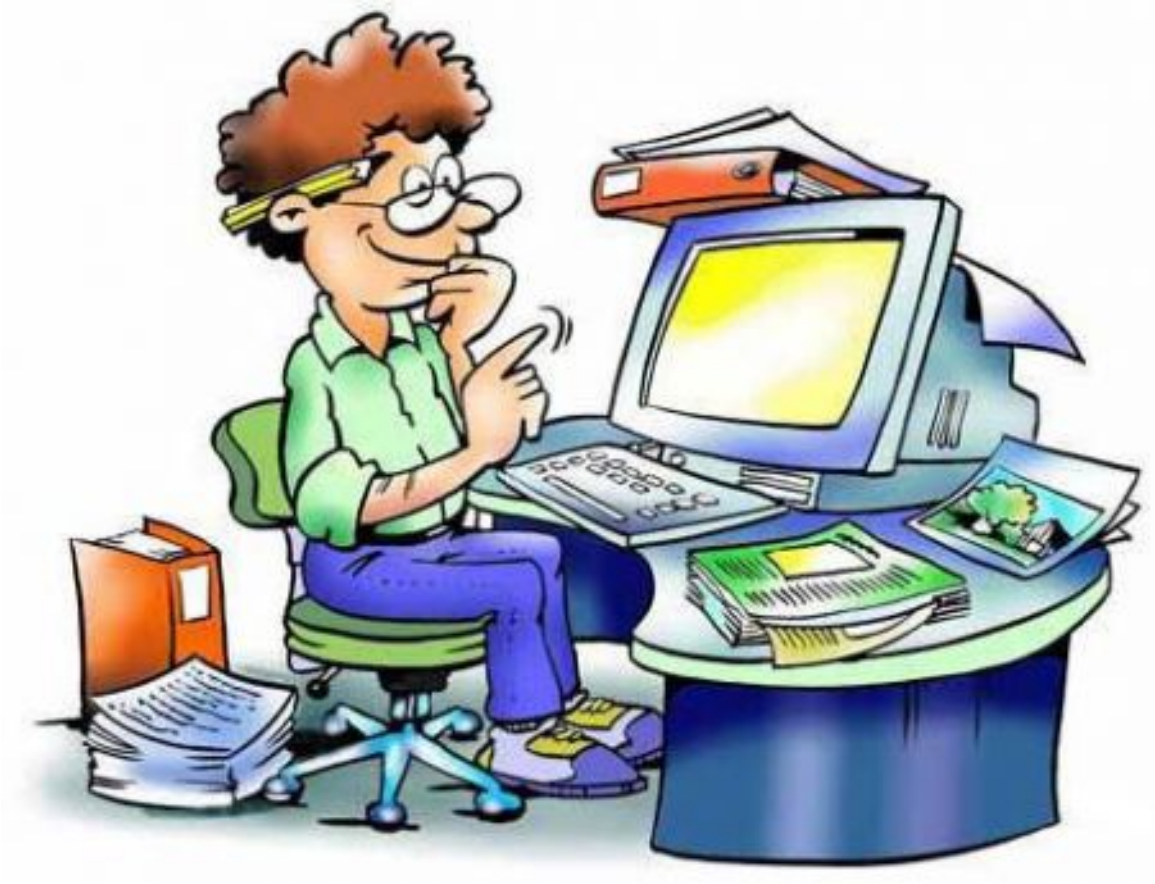
```
$ nohup command-name &
```

htop

- Per visualizzare i programmi in esecuzione (anche in background) si può usare il comando `top`.
  Tuttavia esiste anche una versione più "friendly", detta `htop`, che mostra in modo esplicito l'utilizzo delle risorse



Esercizi



Esercizio

- Completare il file segnali.c aggiungendo gli handler per i segnali:

SIGINT
SIGKILL
SIGHUP
SIGTERM