`pgrep`

Stampa i PID di tutti i processi attivati a partire da una espressione regolare estesa.

```
man pgrep
pgrep '^ba.*$'
pgrep -n bash # PID del processo più recente con "bash"
pgrep -o bash # PID del processo meno recente con "bash"
pgrep -l bash # PID e noe dei processi con "bash"
```

`pstree`

Stampa una rappresentazione compatta dell'albero dei processi.

```
pstree | less -Mr
pstree -a # nomi e argomenti processi in exe
pstree -c # esplicito
pstree -p # mostra i PID
pstree -H PID # evidenzia il ramo da init a PID
```

`/usr/bin/time`

Misura il tempo di completamento ed il consumo delle principali risorse di sistema.

```
man time
/usr/bin/time -f "%E" ./test # tempo completamento
/usr/bin/time -f "%E %c %w" ./test # anche numero di cambi di contesto e richieste di I/O bloccanti
```

`ps`

Fornisce una “istantanea” del consumo di risorse di tutti i processi.

```
man ps
ps ax # stampa tutti i processi attivi
ps fax # vista ad albero (forest) di tutti i processi attivi
ps faxl # vista estesa (più campi)
ps faxl | less -Mr ./test
```

`top`

Monitor periodico dei processi.

```
man top
top
```

`pidstat`

Strumento di monitoraggio delle risorse consumate da singoli processi.

```
pidstat -u # mostra “one shot” l'utilizzazione per ciascuno dei processi attivi finora a partire
dall'accensione del PC.
pidstat -u 1 # mostra ogni secondo l'utilizzazione di CPU per ciascuno dei processi attivi nell'ultimo
secondo.
pidstat -u -p 4000 # mostra ogni secondo l'utilizzazione di CPU per il processo avente PID 4000
pidstat -u -C "bash|terminal" 1 # monitora l'uso CPU di "bash" o "terminal"
pidstat -u -p $(pgrep cpubound) 1
```

Virtual file system

`mount`
Gestisce il processo di montaggio e smontaggio manuale di un file system.

```
mount # opzioni
```

```
mountpoint / # controlla se una data directory sia un mountpoint oppure no.
```

`proc`
File system che contiene statistiche sui componenti hardware e software del kernel.

```
man proc
/proc/<PID> # contengono le info sul processo
```

Riposizionamento

```
apropos -s2,3 reposition # otteniamo
    lseek() # sys call per riposizionare un file
    fseek() # lib function per riposizionare uno stream
```

Un **file hole** è una porzione di un file che non è effettivamente memorizzata su disco, ma che viene trattata come se contenesse zeri quando il file viene letto. Questo avviene grazie a una tecnica chiamata **sparse file** (file sparsi).

```
dd if=/dev/zero of=file-con-hole bs=1M count=10 # crea file grande di zeri
ls -l file-con-hole # dimensione logica del file
ls -s file-con-hole # blocchi allocati dal file system
```

Si compila:

```
#include <sys/types.h>  /* mode_t */
#include <fcntl.h>  /* S_IRUSR, ... */
#include <errno.h>  /* errno, perror() */
#include <unistd.h> /* read(), lseek(), write() */
#include <stdlib.h> /* exit() */
#include <stdio.h>  /* printf() */

#define OFFSET  1024 * 1024 * 10    /* 10 MB */

int main(int argc, char *argv[]) {

    const char file_to_read[] = "file-con-hole";
    char byte = 'a';
    int fd, flags;
    ssize_t ret;
    mode_t mode;
    off_t offset;

    /* apriamo il file in lettura e scrittura */
    flags = O_CREAT | O_RDWR;
    mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    if ((fd = open(file_to_read, flags, mode)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /*
     * Riposizioniamo l'offset oltre la fine
     * del file. Usiamo il flag SEEK_END.
     */

    /* Riposizioniamo l'offset del file */
    if ((offset = lseek(fd, (off_t) OFFSET, SEEK_END)) == -1) {
        perror("lseek");
        exit(EXIT_FAILURE);
```

```
    }

    /* scriviamo un byte alla nuova posizione */
    ret = write(fd, (void *)&byte, sizeof(char));
    if (ret != sizeof(char)) {
        perror("write: non sono riuscito a scrivere un char");
        exit(EXIT_FAILURE);
    }

    if ((close(fd) == -1)) {
        perror("close");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

```
ls -l file-con-hole # 21 MB
# 10MB partenza, 10MB salto, 4KB per memorizzare 'a'
ls -s file-con-hole
```

Buffering

"Monitor dei poveri":

```
mpstat 1 | tr -s " " | cuty -f3 -d " "
```

! NON FUNZIONA
Analizzando il file `tr.c` si scopre che l'I/O è bufferizzato con record logici `BUFSIZ` di 32KB.
-> si imposta il buffer in modalità **line oriented**(`-oL`):

```
mpstat 1 | stdbuf -oL tr -s " " | cut -f3 -d " "
```

Collegamenti

`link()` crea un **hard link** ad un file o directory.
`syslink()` crea un **soft link** ad un file o directory.

```
man 2 link
man 2 syslink
```

`unlink` rimuove un **hard link**.

Almeno due processi hanno aperto lo stesso file. uno dei processi cancella il file.
Il link hard al file sparisce (non può essere più acceduto da nuovi processi). i descrittori di file validi al momento della cancellazione non sono alterati (chi aveva aperto il file prima può continuare ad usarlo).

```
/proc/PID/io
grep write_bytes /proc/*/io # per sapere i bytes scritti
grep write_bytes /proc/*/io | sort -nrk 2 | head -n 30
watch -n 1 -d 'grep write_bytes /proc/*/io | sort -nrk 2 | head -n 30'
```

Misurazione contenuto

`df` riporta il consumo complessivo di ciascun file system attivo.

```
man df
df -f # stampa il report il modo leggibile
```

`du` mostra il consumo di spazio di file e cartelle.

```
man du
du file
du -h file
du -s . # mostra il consumo complessivo della dir attuale
```

Ricerca delle directory più voluminose:

```
for d in $(ls); do du -hs $d; done 2> /dev/null
```

Si entra nella cartella più voluminosa e si ripete, ...

Numero richieste I/O

```
/usr/bin/time ls -lR /
/usr/bin/time dd if=/dev/zero of=a bs=1M count=10
```

Si scelga un processo terminale a caso e si stampi la sua tabella dei file aperti:

```
pgrep -n gnome-terminal
lsof -p PID
```

Tabella file aperti:

```
man lsof
lsof -u andreoli
lsof -c bash
lsof /home/andreoli
lsof -a -u andreoli -c bash
lsof -p 7606
lsof -L1
```

Allocazione dinamica - HEAP

`sbrk()`

```
man sbrk
```

Si esegua più volte `./sbrk`, L'indirizzo del program break dovrebbe cambiare sempre.
Come misura di protezione contro gli attacchi di tipo buffer overflow, il kernel implementa la tecnica di **Linear Address Space Randomization** (**LASR**).
-> Ad ogni esecuzione, l'indirizzo di partenza del program break è generato dinamicamente.
Disattivazione LASR:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Riattivazione:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

`brk()`

```
man brk
```

Le chiamate `brk()` e `sbrk()` provocano allocazioni per pagine. Scrivere fuori area non implica la ricezione di un segmentation fault. Una volta liberata l'area di memoria virtuale (tramite diminuzione del program break) i frame fisici sono restituiti al kernel.
-> Se si prova a scrivere su tale area, si ottiene un segmentation fault.

Allocazione dinamica - MMAP

L'allocazione dinamica di memoria può avvenire anche tramite la creazione di una mappa di memoria con le chiamate di sistema seguenti.

- `mmap()`: crea la mappatura.
- `munmap()`: distrugge la mappatura.
  La mappatura può essere:
- **Anonima**: l'area di memoria virtuale non è associata ad alcun file. Se riferita, viene associata a frame fisici.
- **Con nome**: l'area di memoria virtuale è associata ad un file. Se riferita, comporta la lettura di una porzione di file o il suo aggiornamento su disco.
  `mmap()`
  La chiamata `mmap()` provoca allocazioni per pagine. Scrivere fuori area non implica la ricezione di un segmentation fault.
  Una volta liberata l'area di memoria virtuale (tramite distruzione della mappatura) i frame fisici sono restituiti al kernel.

Allocazione dinamica - MALLOC

L'allocatore standard della libreria del C usa due funzioni di libreria.

- `malloc()`: prenotazione di aree lineari di memoria.
- `free()`: restituzione di aree lineari di memoria.
  Allocazioni piccola: si gestisce una lista di blocchi liberi all'interno dell'heap.
  Allocazioni grandi: si crea una nuova mappatura anonima.
  Per allocazioni $< 128KB$, `malloc()` usa l'heap e mantiene una lista doppiamente collegata di aree libere. L'invocazione di `malloc()` provoca, come prima cosa, la ricerca di un blocco libero sufficientemente grande nella lista. Nel caso in cui sia trovato un blocco disponibile, viene ritornato quello.
  Quando la memoria è liberata tramite `free()` il blocco allocato è reinserito nella lista. I primi byte del blocco (ora libero) sono usati per memorizzare i collegamenti all'elemento libero successivo e precedente.

Misurazione

`free` mostra informazioni sulla memoria a disposizione e sulla sua occupazione.

```
man free
free -h -s 1 # stampa leggibili ogni secondo
```

Campi:

- **Buffers**: Misura il consumo dovuto al buffering dei blocchi nella page cache.
- **Cached**: Misura il consumo dovuto all'uso degli oggetti SLAB.
  Si provi ad annientare tutte le cache (richiede i diritti di root):

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Le prestazioni della macchina dovrebbero calare nettamente.
Buffers aumenta più rapidamente di **Cached**.

Misurazione demand paging

`vmstat` fornisce informazioni specifiche sulle prestazioni del gestore della memoria virtuale.

Misurazione page fault

`sar` fornisce informazioni specifiche sull'attività di page fault.

```
sar -B 1
```

Distruggere tutte le cache, se possibile. Eseguire il seguente comando:

```
lowriter
```

In un altro terminale, monitorare l'attività di paginazione. Quali fault si riscontrano all'inizio?
Si riscontrano major fault per la lettura del programma e delle librerie condivise da disco.