# Sistemi Operativi, Secondo Modulo
# A.A. 2016/2017
# Testo del Primo Homework

Emanuele Gabrielli, Igor Melatti

## Come si consegna

Il presente documento descrive le specifiche per l'homework 1. Esso consiste in 3 esercizi, per risolvere i quali occorre scrivere 3 files che si dovranno chiamare `1.sh` (soluzione del primo esercizio), `2.sh` (soluzione del secondo esercizio) e `3.awk` (soluzione del terzo esercizio). Per consegnare la soluzione, seguire i seguenti passi:

1. creare una directory chiamata `so2.2016.2017.1.matricola`, dove al posto di `matricola` occorre sostituire il proprio numero di matricola;
2. copiare `1.sh`, `2.sh` e `3.awk` in `so2.2016.2017.1.matricola`
3. creare il file da sottomettere con il seguente comando: `tar cfz so2.2016.2017.1.matricola.tgz {1..3}.*`
4. andare alla pagina di sottomissione dell'homework `151.100.17.205/upload/index.php?id_appello=20` e uploadare il file `so2.2016.2017.1.matricola.tgz` ottenuto al passo precedente. **Attenzione:** il suddetto link è raggiungibile solo da indirizzi Sapienza; pertanto, o siete in uno qualsiasi dei laboratori Sapienza, oppure potete settare una VPN come descritto all'URL `https://web.uniroma1.it/sbs/accedi-da-casa/accedi-da-casa-con-bixy#BIXY_info`.

## Come si auto-valuta

Per poter autovalutare il proprio homework, è necessario installare VirtualBox (`https://www.virtualbox.org/`), creare una macchina virtuale da 32 bit ed indicare come disco di tale macchina virtuale quello corrispondente a Lubuntu 14.04-3, come scaricabile da `http://www.osboxes.org/lubuntu/`. È necessario installare gawk, in quanto in questa distribuzione di Lubuntu c'è invece mawk. Per farlo, è sufficiente dare i seguenti comandi: `sudo apt-get update && sudo apt-get upgrade && sudo apt-get install gawk` (rispondere NO alla domanda sul passare alla versione successiva di Lubuntu).

Si consiglia di configurare la macchina virtuale con NAT per la connessione ad Internet, e di settare una "Shared Folder" (cartella condivisa) per poter facilmente scambiare files tra sistema operativo ospitante e Lubuntu. Si consiglia inoltre di installare le "Guest Additions" nel seguente modo: dapprima dare il comando `sudo apt-get install dkms` da un terminale all'interno di Lubuntu, poi scegliere "Insert Guest Additions CD Image" dal menu di VirtualBox, e poi di nuovo da terminale di Lubuntu scrivere `cd /media/osboxes/VBOXADDITIONS*; sudo ./VBoxLinuxAdditions.run`. Infine, riavviare Lubuntu.

All'interno di tale macchina virtuale, scaricare il pacchetto per l'autovalutazione (*grader*) dall'URL `http://twiki.di.uniroma1.it/pub/SO/SO1213AL/SistemiOperativi12CFUModulo220162017/grader.1.tgz`, e copiarlo in una directory con permessi di scrittura per l'utente attuale. All'interno di tale directory, dare il seguente comando:

```
tar xfzp grader.1.tgz && cd grader.1
```

È ora necessario copiare il file `so2.2016.2017.1.matricola.tgz` descritto sopra dentro alla directory attuale (ovvero, `grader.1`). Dopodiché, è sufficiente lanciare `grader.1.sh` per avere il risultato: senza argomenti, valuterà tutti e 3 gli esercizi, mentre con un argomento pari ad $i$ valuterà solo l'esercizio $i$ (in quest'ultimo caso, è sufficiente che il file `so2.2016.2017.1.matricola.tgz` contenga solo l'esercizio $i$).

# Esercizio 1

## Antefatto

In un cluster ci sono $n$ nodi tutti uguali, all'incirca corrispondenti a dei computer fissi come quelli del laboratorio. Tuttavia, tali nodi sono molto potenti: ciascuno di essi ha $c$ cores e $M$ kB di RAM, con $c$ ed $M$ più grandi rispetto ai computer del laboratorio. Pertanto, potendo usare l'intero cluster, si potrebbero lanciare contemporaneamente (in parallelismo perfetto) $nc$ comandi batch (ovvero, non interattivi). Tuttavia, i cluster sono raramente a disposizione di una sola persona. Nel cluster che interessa per questo esercizio, occorre rispettare le seguenti regole:

- si può fare login su un solo nodo apposito;
- si può usare la Bash, interattivamente o con script, in modo tale che ciascun comando non prenda più di 20 minuti;
- per computazioni importanti, che richiedano più nodi e più cores, è necessario usare il PBS (Portable Batch System): si scrive uno script Bash che dichiara all'inizio (sotto forma di commenti speciali) le risorse richieste, e poi lancia i vari comandi per le computazioni che interessano. Una volta scritto tale script, lo si lancia con il comando `qsub script`.

Per quello che riguarda questo homework, è sufficiente sapere che:

- per richiedere $m$ nodi, ciascuno con $d$ cores, occorre usare la seguente linea nello script PBS: `#PBS -l select=`$m$`:ncpus=`$d$`:mpiprocs=`$d$
- se occorre richiedere $m$ nodi con $d$ cores, più un nodo addizionale con solo $d' < d$ cores (questo può succedere se il numero totale di computazioni da richiedere non è un multiplo di $d$), allora la riga di cui sopra va corretta nel seguente modo: `#PBS -l select=`$m$`:ncpus=`$d$`:mpiprocs=`$d$`+1:ncpus=`$d'$`:mpiprocs=`$d'$.
  Nota bene: $m$ può anche essere 0 (se occorre il numero totale di computazioni da richiedere è minore di $d$)
- per specificare che l'account del progetto da cui scalare le ore utilizzate è $A$, occorre usare la seguente linea nello script PBS: `#PBS -A` $A$
- per specificare che l'intera computazione richiederà $h$ ore, $m$ minuti e $s$ secondi, occorre usare la seguente linea nello script PBS: `#PBS -l walltime=`$h$`:`$m$`:`$s$
- le specifiche di cui sopra vanno tutte all'inizio dello script, prima di ogni altro comando Bash
- volendo lanciare i comandi bash $c_1, \ldots, c_{md}$, si usa il seguente comando singolo: `mpirun --hostfile $PBS_NODEFILE -np 1` $c_1$ `: ... : -np 1` $c_{md}$

## Esercizio vero e proprio

Scrivere uno script `1.sh` con i seguenti argomenti (nell'ordine dato):

1. nome dell'account $A$ del progetto;
2. numero massimo di nodi $n$ per script PBS;
3. numero massimo di cores per nodo $c$;
4. nome della directory $D$ di partenza;
5. nome della directory $E$ di output.

Se uno dei suddetti input manca (o ve ne sono in eccesso), l'output dovrà essere semplicemente la scritta `Usage:` $s$ `account num_nodi num_cores input_dir output_dir` su standard error (dove $s$ è il nome dello script), con exit status 10. Se la directory di input non esiste, oppure non ha entrambi i permessi di lettura ed esecuzione per l'utente attuale, l'output dovrà essere semplicemente la scritta `Directory` $D$ `either does not exist or does not have read/execute permissions` su standard error, con exit status 100. Se invece la directory di output non esiste la directory $E$ andrà creata dallo script. Qualora la creazione non vada a buon fine, l'output dovrà essere semplicemente la scritta `Impossible to create directory` $E$ su standard error, con exit status 200. Infine, se la directory $E$ esiste oppure si è riusciti a crearla, ma non è possibile creare file al suo interno, aggiungere i corrispondenti permessi ad $E$. Se l'aggiunta dei permessi non va a buon fine, l'output dovrà essere semplicemente la scritta `Impossible to change permissions for directory` $E$ su standard error, con exit status 150.

Il sottoalbero di directory radicato in $D$ contiene un insieme di script (qui per script si intende un file il cui nome finisca con `.sh` o `.script`), per i quali va creato un opportuno insieme di file PBS, in modo da poterli poi lanciare con `qsub` tutti insieme. Tuttavia, non tutti gli script in $D$ sono da considerare, ma solo quelli per i quali esiste un link, simbolico o no (tale link deve essere sempre contenuto nel sottoalbero di directory radicato in $D$). Si immagini che la fase di etichettamento degli script sia stata effettuata da utenti diversi in momenti diversi, e pertanto tali link potrebbero sia trovarsi nella stessa directory dello script cui fanno riferimento, sia in un'altra. Come regola, vale che gli hard link hanno il nome che termina in `.ln` oppure `.lnk`, mentre i link simbolici possono avere qualsiasi nome. Inoltre, dovranno essere scartati tutti gli script che non hanno tutti i permessi di esecuzione (in tutte e 3 le terne di permessi).

Ciascun file $f$ selezionato secondo le regole di cui sopra contiene una computazione da lanciare su un singolo core, il cui walltime è stimato nel file $f$`.txt`: una singola riga formattata come $h$`:`$m$`:`$s$. Qualora tale file $f$`.txt` non esista, allora lo script $f$ non va selezionato, neanche se è referenziato da un qualche link come descritto sopra.

L'output dello script `1.sh` richiesto da questo esercizio dev'essere un insieme di script PBS, da mettere tutti dentro $E$. Nel dettaglio, dovrà esserci un insieme di script $E$`/pbs.`$h$`.`$m$`.`$s$`.`$i$`.sh` (con $i = 0, 1, \ldots, l_{h,m,s}$) per ogni diverso

valore di $h$:$m$:$s$ nei vari file $f$`.txt`. Pertanto, ogni singolo $E$`/pbs.`$h$`.`$m$`.`$s$`.`$i$`.sh` deve lanciare un certo numero di script $f$ aventi walltime stimato di $h$:$m$:$s$. Per lanciare uno script $f$, occorrerà usare il comando `bash` $f$, e usare per $f$ il path relativo più breve che inizia con `./` (a partire, ovviamente, dalla directory dalla quale viene lanciato lo script `1.sh`). Se uno script $f$ è puntato da più di un link, dovrà comparire una volta sola. Ognuno degli script PBS può allocare al massimo $n$ nodi con $c$ cores ciascuno, pertanto $l_{h,m,s} = \lceil \frac{p_{h,m,s}}{nc} \rceil$, dove $p_{h,m,s}$ è il numero totale di file $f$`.txt` che hanno come valore $h$:$m$:$s$. Ovviamente, il walltime da settare è proprio $h$:$m$:$s$. Infine, per ogni insieme di script $E$`/pbs.`$h$`.`$m$`.`$s$`.`$i$`.sh`, si considerino tutti gli script che vengono lanciati dal comando `mpirun`: tali script dovranno essere ordinati dal più vecchio al più recente (rispetto all'mtime).

Attenzione: non è permesso usare Python, Java, Perl o GCC. Lo script non deve scrivere nulla sullo standard error, a meno che non si tratti di un errore nelle opzioni da riga di comando come descritto sopra. Per ogni test definito nella valutazione, lo script dovrà ritornare la soluzione dopo al più 10 minuti.

## Esempi

Da dentro la directory `grader.1`, dare il comando `tar xfzp all.tgz input_output.1 && cd input_output.1`. Ci sono 6 esempi di come lo script `1.sh` può essere lanciato, salvati in file con nomi `inp_out.`$i$`.sh` (con $i \in \{1, \ldots, 6\}$). Per ciascuno di questi script, la directory di input è `inp.`$i$, e la directory con l'output atteso è `check/out.`$i$.

# Esercizio 2

Scrivere uno script bash con i seguenti argomenti (nell'ordine dato):

1. lista di numeri di PID $p$, separati da `|`
2. numero di bytes $b$
3. walltime $w$ in secondi;
4. intervallo di campionamento $c$

Se uno dei suddetti input manca (o ve ne sono in eccesso), l'output dovrà essere semplicemente la scritta `Usage:` $s$ `pids bytes walltime sampling` su standard error (dove $s$ è il nome dello script).

Lo script dovrà monitorare i processi in $p$. Se uno dei processi monitorati supera $b$ bytes come dimensione totale (contando sia la parte su RAM che quella eventualmente swappata su disco), oppure ha un elapsed time che supera $w$ secondi, va killato simulando con un segnale la pressione di CTRL+C. Il controllo su tali condizioni va effettuato ogni $c$ secondi. Il controllo sulla dimensione del processo va effettuato solo se $b > 0$, e analogamente il controllo sull'elapsed time va fatto solo se $w > 0$. Lo script, se non riscontra errori, deve rimanere in esecuzione finché non trova un file regolare `stop_it.txt` sulla current working directory.

Attenzione: non è permesso usare Python, Java, Perl o GCC. Lo script non deve scrivere nulla sullo standard error, a meno che non si tratti di un errore nelle opzioni da riga di comando come descritto sopra. Per ogni test definito nella valutazione, lo script dovrà ritornare la soluzione dopo al più 10 minuti.

## Esempi

Da dentro la directory `grader.1`, dare il comando `tar xfzp all.tgz input_output.2 && cd input_output.2`. Ci sono 6 esempi di come lo script `2.sh` può essere lanciato, salvati in file con nomi `inp_out.`$i$`.sh` (con $i \in \{1, \ldots, 6\}$). Per ciascuno di questi script, la directory `inp.`$i$ contiene uno script `main.sh` e degli altri file necessari per lanciare `2.sh` e controllare che funzioni correttamente. La directory `check/out.`$i$ contiene i file con l'output atteso.

# Esercizio 3

Scrivere uno script `gawk` che prenda in input 2 files CSV (Comma Separated Value, codificati come testo semplice), in cui ciascuna riga contiene 2 campi separati dalla virgola: il primo campo contiene delle date nel formato `YYYY-MM-DD HH:MM:00`, mentre il secondo campo è un numero razionale. Il numero razionale può essere sia in virgola fissa che in virgola mobile. Più precisamente, avrà uno di questi formati (senza spazi intermedi):

- $a_1.b_1 \dots b_m * 10\hat{}sc_1 \dots c_k$, con $k > 1$, $s \in \{+, -\}$ e $0 \leq a_1, b_i, c_j \leq 9$, ma se $a_1 = 0$ allora anche tutti i $b_i$ e tutti i $c_i$ sono 0 (virgola mobile);
- $a_1 \dots a_n.b_1 \dots b_m$, con $(n > 0 \vee m > 0)$ e $0 \leq a_i, b_j, c_l \leq 9$, ma se $n > 1$ allora $a_1 \neq 0$ (virgola fissa);
- sia nel caso della virgola fissa che della virgola mobile, se $m = 0$ allora il `.` può essere omesso.

Nel formato della data, è possibile (e corretto) che mesi, giorni, ore e minuti abbiano (o non abbiano) degli zeri a sinistra (ad esempio: `2017-00002-1 013:0015:00`). Assumere che le date, se sono scritte con il formato corretto di cui sopra, siano sempre esistenti (ad esempio, non ci sarà mai un 30 febbraio); tuttavia, è possibile che siano sbagliate le ore o i minuti (ad esempio, 24:00:00 o 01:65:00). Se una riga non rispetta le regole di cui sopra, occorre scartarla.

Il primo file contiene i consumi elettrici di una casa in kWh, presi ogni $k \in \mathbb{N}$ minuti consecutivi (con $k$ fisso lungo l'intero file), mentre il secondo file ha i prezzi in EUR/kWh, presi ogni ora (sempre consecutivamente). Qualche riga può mancare sia nel primo che nel secondo file (o non essere data correttamente, come detto sopra): assumere che il consumo nel primo caso ed il prezzo nel secondo siano 0. Inoltre, entrambi i file cominciano con i minuti a 0. Sapendo che $k$ divide 60, lo script deve dare in output, per ogni riga $r$ formattata correttamente del primo file, una riga $r'$ in cui il primo campo è uguale al primo campo di $r$, e il secondo campo contiene il costo in EUR sostenuto nell'attuale intervallo di tempo (arrotondato a 4 cifre dopo la virgola). Alla fine, un'ulteriore riga dovrà mostrare il solo costo totale in EUR nell'intero periodo, arrotondato ad una cifra dopo la virgola.

Infine, lo script dovà scrivere sul file il cui nome è salvato nella variabile `filename` tutte le righe che non erano formattate correttamente.

Attenzione: Lo script non deve scrivere nulla sullo standard error. Per ogni test definito nella valutazione, lo script dovrà ritornare la soluzione dopo al più 10 minuti.

## Esempi

Da dentro la directory `grader.1`, dare il comando `tar xfzp all.tgz input_output.3 && cd input_output.3`. Ci sono 3 esempi di come lo script

`3.awk` può essere lanciato, salvati in file con nomi `inp_out.`$i$`.sh` (con $i \in \{1, \ldots, 3\}$). Per ciascuno di questi script, la directory `inp.`$i$ contiene i 2 file CSV. La directory `check/out.`$i$ contiene i file con l'output atteso.