# Sistemi Operativi, Primo Modulo
# A.A. 2017/2018
# Testo del Secondo Homework

Igor Melatti

## Come si consegna

Il presente documento descrive le specifiche per l'homework 2. Esso consiste in 2 esercizi, per risolvere i quali occorre scrivere 2 programmi Python (versione 3) che si dovranno chiamare `program01.py` (soluzione del primo esercizio) e `program02.py` (soluzione del secondo esercizio). Per consegnare la soluzione, seguire i seguenti passi:

1. creare una directory chiamata `so1.2017.2018.2.matricola`, dove al posto di `matricola` occorre sostituire il proprio numero di matricola;
2. copiare `program01.py` e `program02.py` in `so1.2017.2018.2.matricola`
3. da dentro quest'ultima directory, creare il file da sottomettere con il seguente comando: `tar cfz so1.2017.2018.2.matricola.tgz program*py`
4. andare alla pagina di sottomissione dell'homework `151.100.17.205/upload/index.php?id_appello=38` (attenzione: il copia-incolla dal PDF potrebbe non funzionare bene) e uploadare il file `so1.2017.2018.2.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`, o anche direttamente `https://web.uniroma1.it/sbs/sites/default/files/configurareProxy_SBS.pdf`.

## 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/SistemiOperativi12CFUModulo1Canale120172018/grader.2.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.2.tgz && cd grader.2
```

È ora necessario copiare il file `so1.2017.2018.2.matricola.tgz` descritto sopra dentro alla directory attuale (ovvero, `grader.2`). Dopodiché, è sufficiente lanciare `bash grader.2.sh` per avere il risultato: senza argomenti, valuterà tutti e 2 gli esercizi, mentre con un argomento pari ad $i$ valuterà solo l'esercizio $i$ (in quest'ultimo caso, è sufficiente che il file `so1.2017.2018.2.matricola.tgz` contenga solo l'esercizio $i$).

# Esercizio 1

Il servizio di prenotazioni della compagnia aerea Voli Amo è implementato su un singolo server centrale $S$, che riceve gli ordini di prenotazione da 3 server intermedi: uno per il Nord Italia $N$, uno per il Centro Italia $C$ e uno per il Sud Italia $M$. Per restrizioni software ed hardware, è necessario fare in modo che, in ogni istante, il server centrale $S$ debba gestire al più $R$ richieste. Inoltre, una volta che uno dei 3 server intermedi (ad esempio, $C$) ha cominciato a mandare le sue richieste, gli altri server (nell'esempio, $N$ ed $M$) devono aspettare che tutte le richieste attuali, più altre eventualmente accodatesi sullo stesso server (nell'esempio, da $C$) siano soddisfatte, prima di sottomettere le proprie.

Scrivere una classe Python 3 che si chiami `Prenotazioni` e che abbia i seguenti metodi:

- costruttore con 1 argomento, cui passare il parametro $R$ di cui sopra
- `gestisci_prenotazioni`, con 3 argomenti:
  - `server`: varrà la stringa `"C"` se la richiesta arriva dal server $C$, `"N"` se arriva dal server $N$, `"M"` se arriva dal server $M$;
  - `info`: una stringa arbitraria
  - `scrivi_prenotazione`: una funzione che manderà la prenotazione al server $S$; tale funzione prende un argomento, che dovrà essere la stringa `info`.

La classe `Prenotazioni` verrà quindi usata come proxy di $S$: i server $N$, $C$ ed $M$ non manderanno le loro richieste direttamente ad $S$ (altrimenti potrebbero non rispettare le specifiche di cui sopra), ma invece chiameranno il metodo `gestisci_prenotazioni` di `Prenotazioni`, passandogli ciascuno il proprio identificativo, le opportune `info` aggiuntive della prenotazione, e la funzione che verrà poi eseguita su $S$.

Usare il package standard di Python `threading`.

## Esercizio 2

Scrivere una classe `ResourceAllocSimulator` che implementi l'algoritmo del banchiere. A tal proposito, `ResourceAllocSimulator` deve avere i seguenti metodi:

- Costruttore, con 2 argomenti: `resources` è una lista che corrisponde al "resource vector", mentre `claim` è una matrice che corrisponde al "claim matrix". La matrice `claim` è indicizzata prima sui processi e poi sulle risorse.
- `alloc_req`, che deve simulare una richiesta da parte di un processo. A tal proposito, prende 2 argomenti: `proc_id` che è il PID del processo che fa la richiesta, e `request` che è una lista che rappresenta la richiesta stessa. Più in dettaglio, `request` è una lista in cui, per ogni risorsa, viene indicato il numero di istanze di quella risorsa delle quali si sta facendo richiesta. Il metodo ritorna una delle seguenti stringhe: `"Error"` se la claim matrix non è rispettata dalla richiesta `request` (ovviamente, tenendo conto delle risorse già allocate), `"Blocked"` se la richiesta ha l'effetto di bloccare `proc_id`, e `"OK"` se la richiesta può essere accordata.
- `complete`, che deve simulare il fatto che un processo termini, rilasciando tutte le risorse che gli erano state accordate in precedenza. A tal proposito, prende un solo argomento: il PID `proc_id` del processo che termina. Ritorna un booleano: vero se il processo dato può terminare (ovvero, se ci sono le risorse disponibili perché il processo `proc_id` possa essere eseguito fino alla fine), falso altrimenti (nel qual caso, questo metodo non deve fare nulla). Come effetto collaterale, tutti i processi blocked che, in seguito alla terminazione di `proc_id` e al rilascio delle risorse da esso detenute, potrebbero terminare, devono passare a ready.
- `get_blocked`: ritorna l'insieme dei PID dei processi blocked
- `get_allocated`: ritorna la matrice "allocation matrix" (organizzata come la matrice `claim`). Se un processo è nel frattempo terminato, la sua riga corrispondente, anziché essere una lista, sarà il solo valore `None`.

Valgono le seguenti assunzioni:

- Non si aggiungono mai nuovi processi; al più, qualche processo attuale può terminare quando si chiama `complete`.
- I PID dei processi vanno da 0 ad $n - 1$, con $n$ numero di processi iniziali; il processo con PID $i$ è quello cui corrisponde la $i$-esima riga di `claim`.