1 Installazione e setup dei simulatori per Windows e macOS

1.1 Premessa fondamentale: architetture e compatibilità

Prima di procedere, è essenziale capire che questi simulatori non sono normali applicazioni installabili con un semplice clic. Entrambi sono nati in ambito accademico per Linux, ma hanno esigenze molto diverse legate alla loro "età".

Questi due simulatori architetturali (Gem5 e MarssX86) nascono e vengono sviluppati nativamente per sistemi Unix-like.

- **macOS** è un sistema certificato Unix. "Parla la stessa lingua" di Linux, quindi può eseguire molti di questi programmi nativamente (direttamente dal terminale), con il solo aiuto di alcune librerie esterne.
- **Windows** ha un'architettura completamente diversa, per cui non può eseguire nativamente questi programmi. Per poterli eseguire è quindi necessario utilizzare delle soluzioni di **virtualizzazione o sottosistemi** per creare l'ambiente Unix necessario.

Bisogna anche considerare che, anche avendo un sistema Linux (o Unix), c'è il problema delle librerie software necessarie per compilare il codice (il "Time Gap").

- **Gem5** è un software **moderno e adattivo**, che viene continuamente sviluppato e aggiornato. Funziona senza problemi sui sistemi operativi moderni: su macOS lo si compila direttamente mentre su Windows si usa WSL con un'installazione Linux recente (Ubuntu 20.04/22.04).
- **MarssX86** è un progetto "legacy" (molto vecchio, fermo ad 2012 circa). Richiede compilatori (GCC vecchi) e librerie che sono state rimosse dai sistemi operativi moderni da anni.
  Nè macOS moderno ne WSL possono eseguirlo facilmente, dunque è necessario creare una **Macchina virtuale** (con VirtualBox) che simuli un computer del 2014 con installato **Ubuntu** 14.04.

1.2 Installazione e setup del simulatore Gem5

Per quanto riguarda l'installazione del simulatore Gem5, la soluzione varia leggermente a seconda del sistema operativo utilizzato.

1.2.1 Installazione e setup di Gem5 su Windows

Per installare il simulatore su Windows la soluzione consigliata è **WSL** (Windows Subsystem for Linux). Questo sottosistema permette di avere un terminale Ubuntu vero e proprio integrato in Windows. Dunque, l'installazione si compone di diversi passaggi:

- ***Passo 1*: attivazione di WSL**
  1. Aprire la **PowerShell** come amministratore;
  2. Digitare `wsl --install`, premere invio e attendere il compleatmento del download;
  3. Al termine, seguire le istruzioni a schermo per creare *username* e *password* Ubuntu.

  Una volta fatto ciò, la situazione iniziale nel terminale dovrebbe essere quella mostrata in figura 1, ed eseguendo il comando `pwd` sarà possibile vedere il proprio posizionamento all'interno del sottosistema WSL.

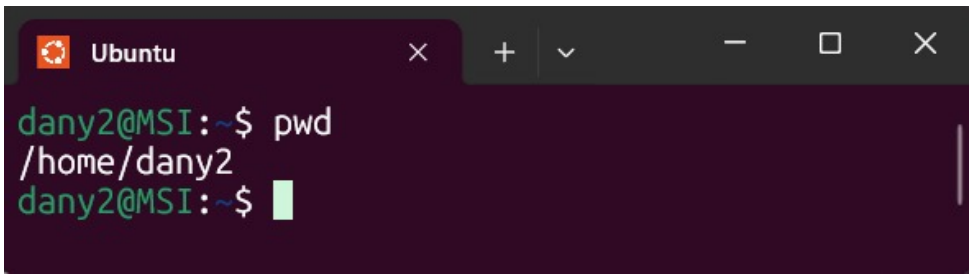

Figure 1: Schermata iniziale del sottosistema WSL

- ***Passo 2*: installazione delle dipendenze**
  Nel terminale di Ubuntu appena aperto incollare questi comandi uno alla volta:

```
//Aggiorna i pacchetti
sudo apt update && sudo apt upgrade -y
```

```
//Installa compilatori, Python e librerie necessarie a Gem5
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev
   libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-
   dev python3-dev python-is-python3 libboost-all-dev pkg-config libpng
   -dev libhdf5-dev libcapstone-dev -y
```

  Note: facendo copia-incolla dei comandi potrebbero verificarsi dei problemi dovuti ai "-" letti dal terminale di Ubuntu come caratteri speciali, in tal caso è consigliato riscriverli a mano.

- ***Passo 3*: scaricare e compilare Gem5**
  Prima di **scaricare** Gem5 è consigliato creare una cartella adatta a poterlo contenere. Quest'operazione viene svolta principalmente perché in futuro dovremo scaricare anche il simulatore architetturale **MarssX86**: così facendo avremo entrambi i simulatori in una cartella apposita.
  Senza spostarsi dalla posizione iniziale `/home/NomeUtente` si va a creare la cartella:

```
mkdir dev_local // Crea la cartella dev_local
cd dev_local // Ci si sposta dentro dev_local per scaricare gem5
```

  Una volta posizionati in `/home/utente/dev_local` è possibile utilizzare il comando `git clone` per scaricare il simulatore gem5 dal repository ufficiale

```
//Clonare il repository del progetto nel proprio workspace
git clone https://github.com/gem5/gem5.git
```

  Quando la clonazione sarà completata, all'interno di `dev_local` sarà presente un'altra cartella nominata "gem5" al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

  Prima di far partire la compilazione è necessario fare alcune **premesse importanti**.
  Utilizzare un calcolatore con 16GB o meno di memoria RAM non basterà: Gem5 è enorme e usa template C++ complessi, ogni core che compila può occupare anche 2-3 GB di RAM. Inoltre, considerando che Windows utilizza parte di questa RAM, la situazione si complica ulteriormente.
  Una possibile soluzione a questo problema, nel caso in cui non si disponesse di un calcolatore

con la quantità di RAM necessaria, è quella di "truccare" WSL per dargli più memoria usando il disco fisso come se fosse RAM (operazione di Swap della memoria):

1. Aprire il blocco note di windows;
2. Incollare esattamente queste righe:

   ```
   [wsl2]
   memory=10GB
   swap=16GB
   ```

   In questo modo stiamo dando a WSL massimo 10GB di RAM vera, ma aggiungiamo 16GB di "memoria lenta" su disco per evitare che la compilazione vada in crash.
3. Salvare il file con nome `.wslconfig` (assicurandosi che non ci sia .txt alla fine!) nella propria cartella utente principale: `C:\Utenti\NomeUtente\.wslconfig`
4. Aprire la PowerShell (sempre in modalità amministratore) e spegnere WSL per applicare la modifica con il comando `wsl --shutdown`

Una volta fatte queste modifiche, è finalmente possibile far partire la compilazione: riaprendo il terminale Ubuntu e posizionandosi in `/home/NomeUtente/dev_local/gem5` deve essere scritto il seguente comando:

```
scons build/ARM/gem5.opt -j 1
```

Tramite di esso la compilazione viene fatta partire utilizzando un solo core della cpu. Nonostante l'utilizzo dello "Swap" con la memoria del disco, la compilazione può essere talmente pesante che utilizzando più di un core per la compilazione si rischierebbe di saturare la RAM mandando in crash la compilazione. È **assolutamente** "**normale**" che i tempi di compilazione varino da 1h a 2h in base alla cpu che si ha a disposizione.

Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando `scons` quì sopra, sostituendo `ARM` con `RISCV`.

- ***Passo 4*: test di funzionamento**
  Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.
  Sempre posizionati in `/home/NomeUtente/dev_local/gem5`, lanciare questo comando:

  ```
  ./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-
     progs/hello/bin/arm/linux/hello
  ```

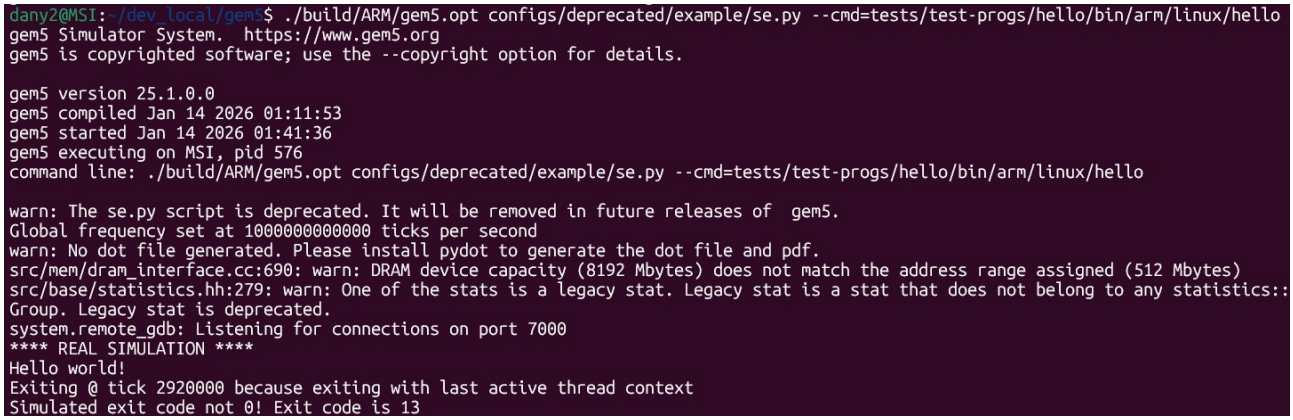

Figure 2: test di simulazione per Gem5

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- "***stats.txt***": Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- "***config.ini***": Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

1.2.2 Installazione e setup di Gem5 su macOS

Sebbene sia teoricamente possibile compilare nativamente su macOS, la gestione delle dipendenze (in particolare Python e Protobuf) risulta spesso instabile a causa degli aggiornamenti di sistema Apple.

Per garantire la **massima riproducibilità** e stabilità, la soluzione raccomandata per macOS è l'utilizzo di un container **Docker** basato su Ubuntu 22.04. Come per WSL, questo approccio permette di avere un ambiente di sviluppo basato su Linux nativo.

- ***Passo 1*: preparazione del container**
  1. Scaricare e installare Docker desktop dal sito ufficiale `https://www.docker.com/`;
  2. È consigliato aprire il terminale posizionandosi nella cartella del proprio apple account come mostrato in figura 3;

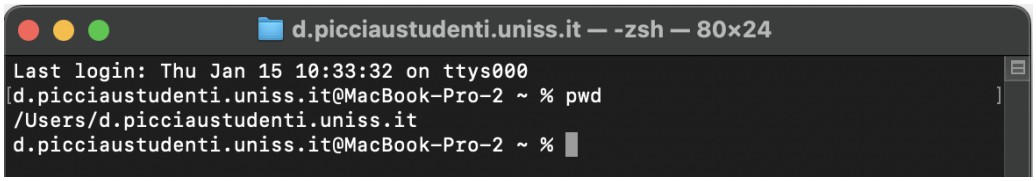

Figure 3: posizionamento iniziale per l'installazione di docker

  3. Creare il proprio workspace che conterrà il simulatore Gem5 e in futuro MarssX86.

```
mkdir dev_local
cd dev_local
```

- ***Passo 2*: installazione delle dipendenze**
  1. All'interno di `/Users/NomeUtente/dev_local`, installare tutte le dipendenze necessarie per la creazione del container di Ubuntu 22.04:

```
docker pull ghcr.io/gem5/ubuntu-24.04_all-dependencies:v25-1
```

  2. Una volta che il workspace è stato creato è possibile far partire il container di Ubuntu (sempre stando posizionati in `/Users/NomeUtente/dev_local`):

```
docker run --volume /Users/NomeUtente/dev_local/gem5:/gem5 --rm -it
   ghcr.io/gem5/ubuntu-24.04_all-dependencies:v24-0
```

     Il completamento di questo comando dovrebbe aprire una riga di comando di questo tipo, ad indicare che siamo all'interno del container Ubuntu.

```
root@{...}:/#
```

- ***Passo 3*: scaricare e compilare Gem5**
  All'interno del container Ubuntu appena creato, è possibile **scaricare** il repository ufficiale tramite il comando `git clone`:

```
root@{...}:/#  git clone https://github.com/gem5/gem5.git
```

Verrà creata una cartella "gem5" al cui interno saranno al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

Come specificato al "***Passo 3***" del capitolo 1.2.1 la compilazione richiede una grande quantità di RAM per cui è richiesto un calcolatore con 16 o più GB di memoria RAM (solo per la compilazione di Gem5!). Dopo alcuni test da noi effettuati è risultato che la compilazione su un calcolatore con 8GB di RAM potrebbe impiegare anche mezza giornata!
Proprio per questo motivo, anche disponendo di 16GB di memoria RAM è necessario andare nella GUI di Docker `impostazioni` $\rightarrow$ `resources` $\rightarrow$ `advanced`, e da qui cambiare le seguenti impostazioni:

- ***Memory limit***: impostiamo la capacità massima di memoria RAM che Docker può utilizzare. È consigliato impostarlo quasi al massimo, (ad esempio, con 16 GB di RAM a disposizione impostarlo a $14/15$) per permettere al sistema operativo (macOS) di "sopravvivere"
- ***Swap***: Invece, tramite questa impostazione, indichiamo quanto spazio di "memoria lenta" prendere dal disco nel caso in cui la RAM vada in saturazione a causa della pesantezza di compilazione.

Una volta che il repository è stato clonato ci si sposta all'interno della cartella "gem5" per far partire la **compilazione del simulatore**, utilizzando 1 o al massimo 2 core per evitare che la compilazione vada in crash a causa della saturazione della RAM a dispozione.

```
// Mi sposto all'interno della cartella gem5
root@1ab1ef08f2f:/# cd gem5

// Compilazione ARM
root@1ab1ef08f2f:/gem5# scons build/ARM/gem5.opt -j1
```

Come per Windows, anche la compilazione su macOS può impiegare da 1h a 2h, ed è "**normale**" vista la mole di file da scaricare.
Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando `scons` quì sopra, sostituendo `ARM` con `RISCV`.

***Passo 4*: test di funzionamento**
Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.
Sempre posizionati in `root@1ab1ef08f2f:/gem5#`, lanciare questo comando:

```
./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-
   progs/hello/bin/arm/linux/hello
```

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- "***stats.txt***": Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- "***config.ini***": Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

1.3 Installazione e setup del simulatore MarssX86

Nonostante sia un processo abbastanza lungo e tedioso la scelta consigliata per effettuare l'installazione del simulatore, sia per Windows che per macOS, è utilizzare Docker.

1.3.1 Installazione e setup di MarssX86 su Windows

L'utilizzo di Docker è la scelta raccomandata per mantenere un flusso di lavoro unificato. Grazie all'integrazione con WSL 2, è possibile avviare l'ambiente "legacy" necessario per MarssX86 direttamente dallo stesso terminale utilizzato per Gem5 (come visto al passo 1 del capitolo 1.2.1), senza dover configurare macchine virtuali esterne pesanti.

- ***Passo 1*: Configurazione (WSL + Docker)**
  1. Scaricare e installare Docker desktop dal sito ufficiale `https://www.docker.com/`;
  2. Aprire docker desktop su Windows;
  3. Andare nelle impostazioni (icona dell'ingranaggio in alto a destra);
  4. Spostarsi nella sezione **resources** e poi in **WSL integration**;
  5. Assicurarsi di attivare la casella "Ubuntu" per abilitare l'integrazione;
  6. Cliccare su "*apply e restart*"

  A questo punto, aprendo il terminale WSL, è possibile verificarne il funzionamento tramite il comando `docker -version`.

- ***Passo 2*: preparazione del container**
  1. Nel terminale WSL spostarsi all'interno della cartella `/home/NomeUtente/dev_local` (come fatto al passo 3 del capitolo 1.2.1), e creare una cartella per il progetto Marss con al suo interno il file di definizione dell'ambiente:

     ```
     mkdir marss
     cd marss
     nano Dockerfile
     ```

     All'interno dell'editor che si aprirà dopo l'ultimo comando (`nano Dockerfile`), incollare il seguente contenuto che definisce un ambiente Ubuntu 18.04 con tutte le dipendenze necessarie.
     Nonostante MarssX86 sia stato scritto in un ambiente Ubuntu 14.04 (risalente al 2012), non è possibile replicare tale ambiente, poiché da molti anni i server hanno cancellato tutti i file. Dunque una possibile strategia è quella di usare un sistema operativo più recente (Ubuntu 18.04) i cui server funzionano perfettamente, ma installare manualmente i "vecchi" compilatori.

     ```
     # Usiamo Ubuntu 18.04 che ha server ancora attivi e stabili
     FROM ubuntu:18.04

     # Aggiorniamo i repository (funzionera' senza errori 404)
     RUN apt-get update

     # Installiamo i tool di base e specificamente GCC 4.8 / G++ 4.8
     # MarssX86 richiede queste versioni antiche per compilare
     RUN apt-get install -y build-essential git scons zlib1g-dev python2.7
         python-minimal g++ \
         vim libsdl1.2-dev gcc-4.8 g++-4.8
     ```

```
# Configuriamo il sistema per usare GCC 4.8 come default assoluto
# Questo "inganna" Marss facendogli credere di essere su un vecchio
   sistema
RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8
   100 && \
    update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8
       100

# Cartella di lavoro
WORKDIR /root/marss

CMD ["/bin/bash"]
```

2. Rimanendo posizionati in `/home/NomeUtente/dev_local/marss`, lanciare il comando di build per l'ambiente Ubuntu. Verrà richiesta una password, inserire quella dell'account Ubuntu come fatto al "***Passo 1***" del capitolo 1.2.1.

```
sudo docker build --no-cache -t marss_env .
```

3. Infine, è possibile creare l'ambiente ubuntu scelto. Servirà la password.

```
sudo docker run -it --name marss_container -v "$PWD":/root/marss
   marss_env
```

Nota importante: l'aggiunda del comando `-v "$PWD":/root/marss` crea un collegamento diretto tra la cartella in cui ci si trova (`/home/NomeUtente/dev_local/marss`) nel disco e la cartella di lavoro "virtuale" del container. Senza di esso non si vedrebbero le modifiche effettuate nel container, costringendo a copiare ogni file che si vuole utilizzare nel disco del computer, tramite comandi aggiuntivi. Ovviamente, il container rimane necessario per compilare/eseguire i file dell'ambiente Ubuntu scelto.

Una volta fatto ciò, si dovrebbe essere riusciti ad entrare nell'ambiente di simulazione, e a schermo dovrebbe vedersi questo:

```
root@{...}:~/marss#
```

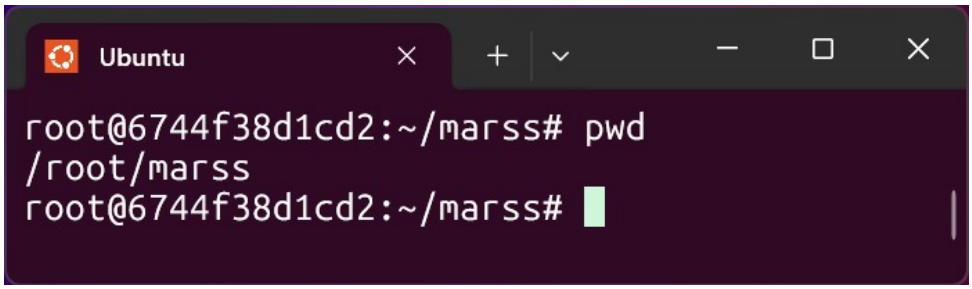

Figure 4: ambiente di simulazione del MarssX86

Inserendo subito il comando `"pwd"` si potra vedere come ci si trova in `/root/marss` come specificato nel Dockerfile.

4. Per **tornare alla bash di Ubuntu** basterà scrivere il comando `"exit"` da qualsiasi posizione, mentre per rientrare nell'ambiente già creato servirà un comando differente da lanciare (posizionati in `/home/NomeUtente/dev_local/marss`):

```
docker start -ai marss_container
```

Invece, per rimuovere il container serve prima sapere il container id. Nell'ambiente Ubuntu (quindi dopo avere fatto "exit") utilizzare il seguente comando:

```
docker ps
```

Subito dopo

```
docker rm <container id>
```

- ***Passo 3*: scaricare e compilare MarssX86**
  Dopo essere riusciti ad entrare dentro l'ambiente di simulazione, bisogna **scaricare il repository** del progetto ufficiale (posizionati in `/root/marss`):

```
root@{...}:~/marss# git clone https://github.com/avadhpatel/marss.git
```

Una volta completata la clonazione, all'interno di `/root/marss` verrà scaricata un ulteriore cartella "marss" contenente i file reltivi al repository.
Anche in questo caso la compilazione risulta abbastanza pesante a causa della quantità di RAM richiesta. Fortunatamente, Docker desktop su Windows eredita da WSL il file `.wslconfig` (***passo 3*** del capitolo 1.2.1) creato per fare in modo che la compilazione non vada in crash.

Prima di **procedere con la compilazione** ci sono alcune problematiche che vanno risolte, relative alle discrepanze tra l'utilizzo di comandi moderni e compilatori vecchi oppure causate da librerie di sicurezza moderne (già presenti nella versione 18.04 di Ubuntu che stiamo utilizzando) che bloccano la compilazione del file finale di compilazione. Queste problematiche sono sorte in fase di compilazione stessa, e avendole individuate, è necessario risolverle per garantire il funzionamento della compilazione.

1. Posizionarsi in `/root/marss/marss/ptlsim` e aprire il file "*SConstruct*" che rappresenta il "cuore" del simulatore PTLsim:

```
root@{...}:~/marss/marss/ptlsim# apt-get install -y nano
root@{...}:~/marss/marss/ptlsim# nano SConstruct
```

2. Una volta all'interno dell'editor di testo nano per il file "*SConstruct*" cercare la riga:

```
env.Append(CCFLAGS = ['-fdiagnostics-color=always'])
```

ed eliminarla o commentarla utilizzando un "#". Per aiutarsi è possibile entrare in modalità ricerca con la combinazione di comandi `ctrl + w`.

Subito sotto questa riga, ne dovrebbe essere presente un'altra di questo tipo:

```
env.Append(CCFLAGS = ' -std=gnu++11 ')
```

la quale deve essere modificata in questo modo:

```
env.Append(CCFLAGS = ' -std=gnu++11 -U_FORTIFY_SOURCE ')
```

In questo modo stiamo dicendo al compilatore di "*disabilitare i controlli di sicurezza moderni per il codice vecchio del repository*".
Premere `ctrl+o` per salvare le modifiche e `ctrl+x` per uscire dall'editor di testo.

3. Tornati nell'ambiente di simulazione bisogna effettuare lo stesso procedimento nel file "*SConstruct*" principale, quindi non quello presente in `/root/marss/marss/ptlsim` ma in `/root/marss/marss/`, quindi:

```
root@{...}:~/marss/marss# nano SConstruct
```

Una volta all'interno dell'editor di testo, bisogna scendere nel file fino a trovare un comando di questo tipo:

```
if int(pretty_printing) :
    base_env = Environment(
            CXXCOMSTR = compile_source_message,
            CREATECOMSTR = create_header_message,
            CCCOMSTR = compile_source_message,
            SHCCCOMSTR = compile_shared_source_message,
            SHCXXCOMSTR = compile_shared_source_message,
            ARCOMSTR = link_library_message,
            RANLIBCOMSTR = ranlib_library_message,
            SHLINKCOMSTR = link_shared_library_message,
            LINKCOMSTR = link_program_message,
            )
else:
    base_env = Environment()
```

Questo blocco serve a creare l'ambiente di compilazione (base_env). Bisogna intercettare quella variabile appena è stata creata e aggiungere il "*flag salvavita*" che disabilita i controlli di sicurezza troppo moderni. Dopo l'else bisogna aggiungere le seguenti righe:

```
...
...

else:
    base_env = Environment()

base_env.Append(CFLAGS = ['-U_FORTIFY_SOURCE'])
base_env.Append(CCFLAGS = ['-U_FORTIFY_SOURCE'])
```

Fatto ciò, nuovamente `ctrl+o` per salvare le modifiche, invio, e poi `ctrl+x` per tornare nell'ambiente di simulazione.

Effettuate queste modifiche è finalmente possibile **compilare** tramite il comando:

```
root@{...}:~/marss/marss# scons -Q C=1 debug=0
```

Come risultato finale si dovrebbe visualizzare un comando di questo tipo: