Universita' degli Studi di Messina
Dipartimento di Matematica e Informatica

Progetto Sistemi Operativi

AllocSim

24 Giugno 2015

Autore:
Vittorio Romeo

Professoressa:
Santa Agreste

http://vittoriomeo.info

http://unime.it

Contents

1. Richiesta 1
2. Algoritmi di allocazione 1
3. Design ed implementazione 2
   - 3..1 Ambiente e tool di sviluppo 2
   - 3..2 Architettura applicazione 2
   - 3..3 Modalità d'uso 6
4. Statistiche 6
5. Diagrammi UML 7
   - 5..1 Class diagram 7
   - 5..2 Activity diagram 7
6. Manuale d'uso 10
   - 6..1 Modalità manuale 10
   - 6..2 Modalità automatica 10
7. Risultati ottenuti 11
8. Link e riferimenti 13

1. Richiesta

Si richiede l'implementazione di un applicativo che simuli tre delle tecniche usate per eseguire l'allocazione della memoria utilizzando una free-list: **first-fit**, **best-fit** e **worst-fit**.

L'applicativo deve prevedere una situazione iniziale (randomica) della memoria e confrontare il risultato finale ottenuto dalle tre tecniche.

Supponendo che ogni confronto costi una unitò, si analizzino le differenze in termini di costo e in termini di frammentazione esterna.

L'applicativo deve effettuare delle simulazioni al fine di esaminare il diverso comportamento delle tre tecniche mostrando ad ogni simulazione lo stato del sistema sia a video che su file di log.

L'applicativo deve essere sviluppato in **Python**.

2. Algoritmi di allocazione

AllocSim implementa i seguenti algoritmi di allocazione dei processi.

- **First-fit**: l'allocatore sceglierà il primo blocco abbastanza grande da poter contenere il processo.
- **Next-fit**: l'allocatore sceglierà il primo blocco (a partire dalla posizione dell'allocazione precedente) abbastanza grande da poter contenere il processo.
- **Best-fit (naive)**: l'allocatore scorrerà tutti i blocchi ed allocherà il processo in quello più piccolo che può contenerlo.
- **Worst-fit (naive)**: l'allocatore scorrerà tutti i blocchi ed allocherà il processo in quello più grande.
- **Best-fit (sorted list)**: l'allocatore terrà traccia dei blocchi in una lista ordinata per la loro grandezza. Partendo dall'inizio della lista sceglierà il primo blocco non occupato che può contenere il processo.
- **Worst-fit (sorted list)**: l'allocatore terrà traccia dei blocchi in una lista ordinata per la loro grandezza. Partendo dalla fine della lista sceglierà il primo blocco non occupato che può contenere il processo.

3. Design ed implementazione

3..1 Ambiente e tool di sviluppo

L'applicazione è stata implementata utilizzando il linguaggio di programmazione **Python 2.6**. L'ambiente di sviluppo utilizzato è **Arch Linux x64**. L'editor utilizzato per la stesura del codice è **Sublime Text 3**.

Il codice del progetto è open-source e disponibile pubblicamente sotto licenza **MIT** su **GitHub** al seguente indirizzo:

`https://github.com/SuperV1234/AllocSim`

Il documento corrente è stato scritto usando LaTeX, un sistema tipografico di alta qualità con numerose caratteristiche che agevolano la stesura di documenti **scientifici** e riguardanti la **programmazione**.

Un piccolo preprocessore LaTeX scritto in **C++14** chiamato **LatexPP** è stato sviluppato ed utilizzato per la creazione di questo documento.

LatexPP permette di usare una sintassi intuitiva che evita la ripetizione di markup per l'highlighting del codice e delle macro.

La preprocessione e compilazione di un documento LaTeX usando LatexPP è stata automatizzata usando il seguente script **bash**.

```
#!/bin/bash

latexpp ./manual.lpp > ./manual.tex
pdflatex -shell-escape ./manual.tex && chromium ./manual.pdf
```

3..2 Architettura applicazione

L'architettura dell'applicazione è stata designata utilizzando **principi OOP** (Object-Oriented Programming) e massimizzando la riusabilità delle classi.

Nella lista seguente sono riportati gli elementi che compongono l'architettura e il modo in cui sono relazionati, insieme a frammenti di codice che mostrano l'interfaccia delle classi più importanti dell'applicativo:

- **Blocco di memoria**: classe Python `Block` - rappresenta porzione dell'area di memoria presente nell'**Allocatore**. Può essere **occupato** o **libero**, ha un **byte di inizio** ed un **byte di fine**.

```
# Class representing a block of memory
class Block:
```

```
    # Constructor
    def __init__(self, mAllocator, mStart, mEnd):
        # Allocator that owns the block
        self.allocator = mAllocator

        # Byte where the memory block begins
        self.start = mStart

        # Byte where the memory block ends
        self.end = mEnd

        # Is the memory block currently occupied?
        self.occupied = False

    # Returns the size of the memory block in bytes
    def getSize(self):
        return self.end - self.start
```

- **Allocatore**: classe Python `Allocator` - rappresenta un'area di memoria contigua che può essere frammentata ed utilizzata per instanziare **processi**. L'area di memoria viene divisa in **blocchi**. Contiene una lista di blocchi ordinata per posizione (in byte), una lista di blocchi ordinata per grandezza (utilizzata per implementare versioni degli algoritmi **best-fit** e **worst-fit** più efficienti), ed una lista di blocchi calcolata a runtime dei blocchi adiacenti liberi (**boundary-tag**). L'allocatore permette di **dividere** un blocco in due in una specifica posizione, di **occupare** o **liberare** la memoria, e di **unificare** i blocchi contigui liberi.

```
# Class representing a memory allocator
class Allocator:
    # Constructor
    def __init__(self, mSize): ...

    # Restores the allocator to its original state
    def reset(self): ...

    # Return a tuple containing the block that includes the byte 'mX' and its index
    def getBlockAt(self, mX): ...

    # Inserts a block in the sorted list, in the correct position
    def insertSorted(self, mX): ...

    # Splits the memory owned by the allocator at the byte 'mX'
    # Returns a tuple containing the two halves in which the block was split and the
    # index of the first half
```

```
18      def splitAt(self, mX): ...
19
20      # Merges all adjacent unoccupied blocks
21      def reclaim(self): ...
22
23      # Free the memory in all the blocks, making them unoccupied
24      def free(self): ...
25
26      # Print an ASCII graph of the state of the allocator
27      def printInfo(self): ...
28
29      # Return a number representing the average fragmentation of the allocator
30      def getFragmentation(self): ...
31
32      # Execute an algorithm and print its result
33      def executeAlgorithm(self, mAlgorithm, mX): ...
34
35      # The "first fit" algorithm simply iterates over all blocks
36      # until it finds a block which is suitable for the desired
37      # memory allocation
38      def insertFirstFit(self, mX): ...
39
40      # The "next fit" algorithm uses the same logic as the "first fit" algorithm,
41      # but starts looking for an unoccupied block at the last block index
42      def insertNextFit(self, mX): ...
43
44      # The "best fit" algorithm iterates over all available blocks
45      # and selects the one that wastes less space for the memory allocation
46      # This algorithm could be improved by keeping a separate list of memory blocks,
47      # ordered by size
48      def insertBestFitNaive(self, mX): ...
49
50
51      # The "worst fit" algorithm iterates over all available blocks
52      # and selects the one that wastes most space for the memory allocation
53      # This algorithm could be improved by keeping a separate list of memory blocks,
54      # ordered by size
55      def insertWorstFitNaive(self, mX): ...
56
57      # The "best fit" algorithm starts from the beginning of the sorted list
58      def insertBestFitSL(self, mX): ...
59
60      # The "worst fit" algorithm starts from the end of the sorted list
61      def insertWorstFitSL(self, mX): ...
```

L'allocatore contiene tutta la logica riguardante l'esecuzione degli **algoritmi** richiesti,

ed anche la logica per la generazione di un **grafico ASCII** che rappresenta lo stato dell'area di memoria.

- **Algorithmi**: contenuti nella classe Python `Allocator` - oltre agli algoritmi richiesti (**first-fit**, **best-fit**, **worst-fit**), sono stati designati ed implementati anche i seguenti: **next-fit**, **best-fit** (con lista ordinata), e **worst-fit** (con lista ordinata). Gli algoritmi agiscono sui **blocchi**, liberi o occupati da **processi**.

- **Simulazione**: gestite da Python tramite le funzioni `runSimulations` and `simulate` - una simulazione, al suo inizio, genera una lista randomica di **processi**, la quale viene testata in N **sottosimulazioni**, le quali eseguono uno degli **algoritmi** sulla medesima lista di processi, restituendo un **set di risultati**.

- **Sottosimulazione**: esegue un **algoritmo** su un set di **processi** generato randomicamente. Pulisce l'**allocatore** dalle sottosimulazioni precedenti ed esegue calcoli statistici per verificare l'efficienza di un algoritmo, riportata in un **set di risultati**.

- **Processo**: classe Python `Process` - rappresenta un processo del sistema operativo utilizzato per le simulazioni automatiche. Ogni processo ha un **tempo di entrata** (momento in cui il sistema prova ad allocare il processo), un **tempo di esecuzione** (numero di unità di tempo necessarie al completamento del processo), un numero di **byte richiesti** per l'esecuzione del processo (che saranno forniti, se possibile, attraverso l'allocazione di un **blocco** libero). I processi randomicamente vengono generati all'inizio di una **simulazione**.

```
# Class representing a single process
class Process:
    # Constructor
    def __init__(self, mID, mStart, mRequired, mMem):
        # Process ID
        self.id = mID

        # Execution start time
        self.start = mStart

        # Time required for execution
        self.required = mRequired

        # Memory required for execution
        self.size = mMem

        # Block being occupied by the process
        self.block = None
```

```
19
20      # Returns 'True' if the process has ended
21      def finished(self):
22          return self.required <= 0
```

- **Set di risultati**: instanziato e stampato alla fine di una **simulazione**. Contiene informazioni riguardo il **numero di comparazioni**, **frammentazione media** e **punteggio totale** di ogni **algoritmo** testato durante la simulazione. Minore il punteggio, più efficiente l'algoritmo.

3..3 Modalità d'uso

L'applicativo può essere eseguito in due modalità: **modalità manuale** e **modalità automatica**.

- **Modalità manuale**: gestita dalla funzione Python `mainManual` - questa modalità permette all'utente di interagire manualmente con l'**allocatore**, eseguendo allocazioni singole e visualizzando tramite un **grafico ASCII** il loro risultato.
- **Modalità automatica**: gestita dalla funzione Python `mainAutomatica` - genera N simulazioni (valore passato da linea di comando, come primo parametro) e le esegue, scrivendo a video e su log i loro risultati.

4. Statistiche

Le statistiche effettuate dall'applicativo sono le seguenti:

- **Confronti effettuati**: per verificare l'efficienza di un algoritmo rispetto ad un altro verrà conteggiato il numero dei confronti effettuati nella ricerca di un blocco adatto ad un processo.
- **Frammentazione esterna media**: ogni iterazione di una simulazione verrà conservata ed accumulata la frammentazione esterna corrente. Alla fine della simulazione la frammentazione accumulata verrà divisa per il numero di iterazioni e mostrata all'utente.

  Durante una singola iterazione la frammentazione media viene calcolata con la formula seguente:

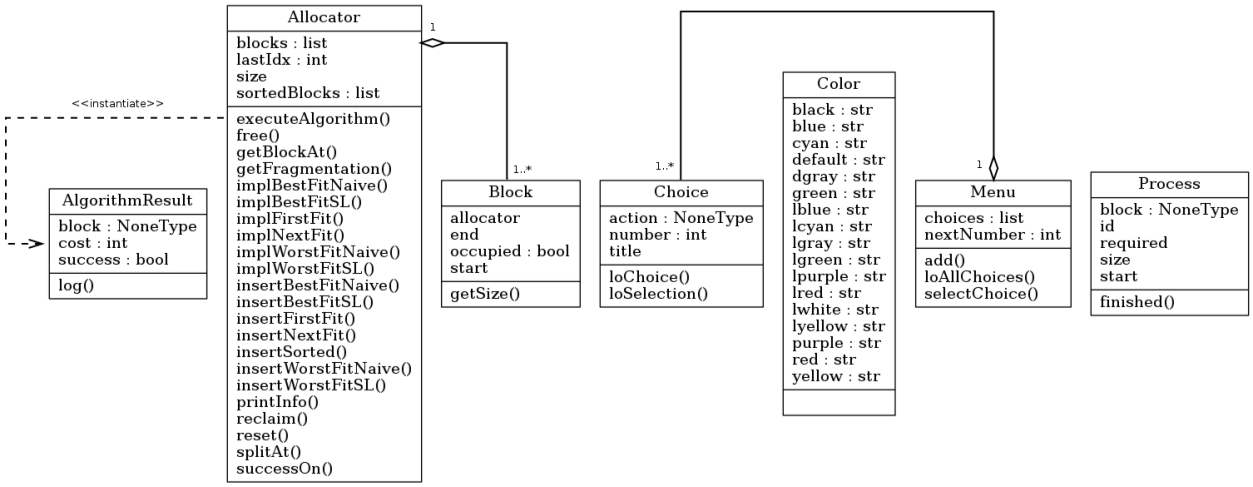
  $externalFragmentation = 1.0 - \frac{biggestFreeBlockSize}{totalFreeMemory}$

5. Diagrammi UML

5..1 Class diagram

Viene qui riportato il **class diagram UML** delle classi contenute nel progetto. Il diagramma è stato realizzato automaticamente dall'applicazione `pyreverse`.

Figure 1: Diagramma UML delle classi.



5..2 Activity diagram

Sono riportati a seguire due **activity diagram**, che mostrano la funzionalità dell'applicativo al suo avvio e il funzionamento dell'algoritmo best-fit.

Figure 2: Diagramma: esecuzione applicativo.

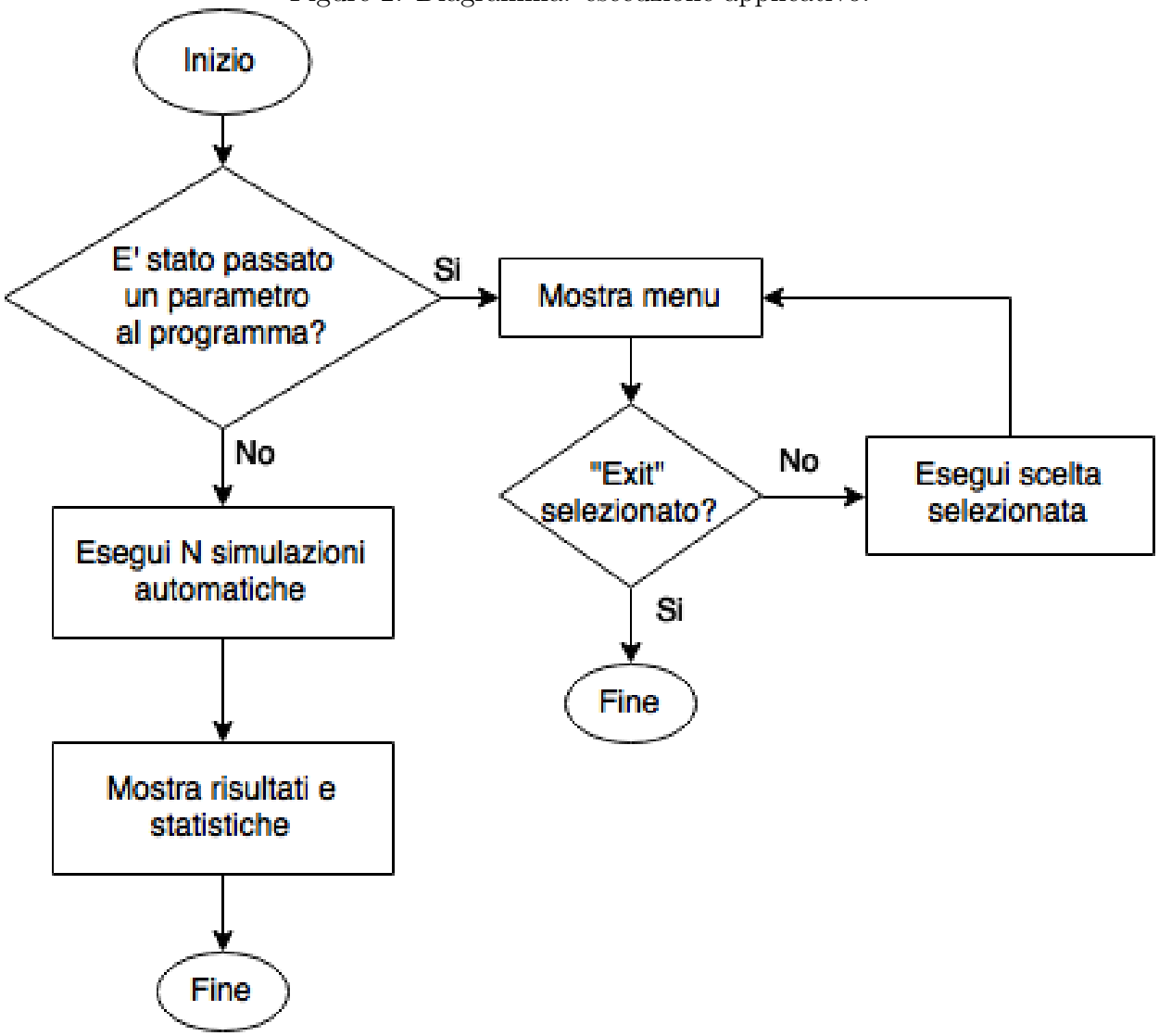
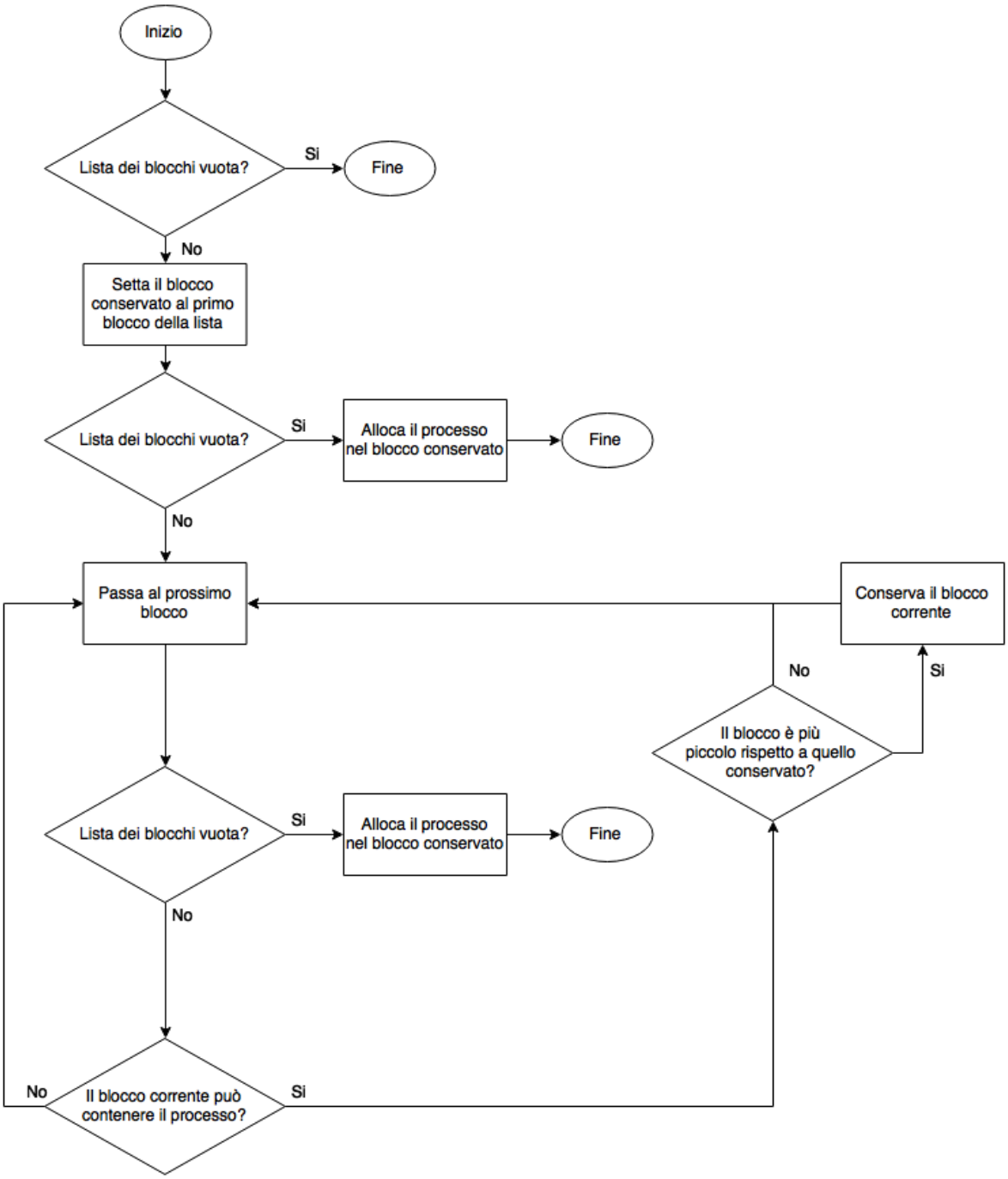


Figure 3: Diagramma: algoritmo best fit.



6. Manuale d'uso

L'applicazione è molto **user-friendly**: semplice da usare e robusta. Gli input forniti dall'utente vengono controllati, ed in caso di input non validi l'utente può ritentare senza causare errori o crash.

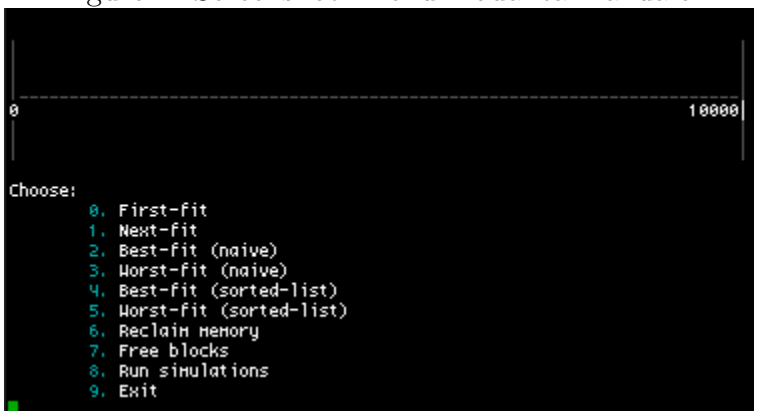
6..1 Modalità manuale

Per avviare l'applicazione in **modalità manuale** è sufficiente lanciarla da linea di comando senza parametri:

```
./project.py
```

Dopo aver avviato il programma, l'utente vedrà il seguente menù:

Figure 4: Screenshot: menù modalità manuale.



L'utente potrà scegliere tra le varie funzioni dell'allocatore e visualizzare su schermo in maniera grafica i loro risultati.

6..2 Modalità automatica

Per avviare l'applicazione in **modalità automatica** è sufficiente lanciarla da linea di comando con il numero di simulazioni desiderato:

```
# Esempio (3 simulazioni)
./project.py 3
```

Dopo aver avviato il programma, le simulazioni partiranno automaticamente ed il loro funzionamento sarà mostrato sia a video che su log.

Figure 5: Screenshot: modalità automatica.

7. Risultati ottenuti

Avendo eseguito 1000 simulazioni ripetutamente ed avendo calcolato la media sui dati statistici restituiti da esse, notiamo che i valori riportati sono in linea con quelli comunemente aspettati per gli algoritmi scelti.

```
TOTAL Results for first-fit:
    Avg. comparisons: 419
    Avg. fragmentation: 37.0976293895

TOTAL Results for next-fit:
    Avg. comparisons: 405
    Avg. fragmentation: 37.227405137

TOTAL Results for best-fit (naive):
    Avg. comparisons: 419
    Avg. fragmentation: 35.3508336604

TOTAL Results for worst-fit (naive):
    Avg. comparisons: 442
    Avg. fragmentation: 37.4901873478

TOTAL Results for best-fit (sorted list):
    Avg. comparisons: 432
    Avg. fragmentation: 35.7433916187

TOTAL Results for worst-fit (sorted list):
    Avg. comparisons: 416
    Avg. fragmentation: 37.0976293895
```

I risultati sono stati calcolati simulando l'entrata ed uscita continua di processi di dimensioni randomiche e durata randomica.

E' possibile notare che, utilizzando questo tipo di simulazione:

- il numero medio di comparazioni e frammentazione su tutte le 1000 simulazioni varia poco in base all'algoritmo. Questo è dovuto alla grandezza e durata randomica dei processi: è a volte possibile che un processo occupi l'intero allocatore per più unità di tempo o che molti processi piccoli e rapidi escano ed entrino continuamente.

- gli algoritmi **first-fit** e **next-fit** hanno risultati estremamente simili. Il **next-fit**, tuttavia, ha in media un numero minore di comparazioni in quanto i blocchi allocati in precedenza vengono subito saltati grazie all'indice conservato che punta al blocco dell'ultimo allocazione.

- l'algoritmo **best-fit** (sia naive che con sorted-list) ha sempre **frammentazione esterna minore** rispetto all'algoritmo **worst-fit** (sia naive che sorted-list). Il numero di comparazioni tra i due non è legato alla scelta dell'algoritmo ma alla generazione randomica dei processi - il **worst-fit** può avere meno comparazioni del **best-fit** e viceversa.

8. Link e riferimenti

Nella stesura di questo documento e nell'implementazione del progetto AllocSim i seguenti riferimenti sono stati utilizzati:

- Operating System Concepts - Silberschatz: `http://os-book.com/`
- Wikipedia: Memory management: `https://en.wikipedia.org/?title=Memory_management`
- Wikipedia: Fragmentation: `https://en.wikipedia.org/wiki/Fragmentation_(computing)`
- ShareLaTeX learn: `https://www.sharelatex.com/learn`
- Wikipedia - Software engineering: `http://en.wikipedia.org/wiki/Software_engineering`
- Sito web UNIME: `http://unime.it`
- Il mio sito personale: `https://vittorioromeo.info`
- LatexPP su GitHub: `https://github.com/SuperV1234/Experiments`
- Documentazione Git: `https://git-scm.com/documentation`
- GitHub: `https://github.com/`
- Wiki di Arch Linux: `https://wiki.archlinux.org/`