Universita' degli Studi di Messina
Dipartimento di Matematica e Informatica

Progetto Sistemi Operativi

AllocSim

24 June 2015

Author:
Vittorio Romeo

Professors:
Santa Agreste

http://vittoriomeo.info

http://unime.it

Contents

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. Design ed implementazione

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:

- **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**.

- **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. 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**.

- **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.

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

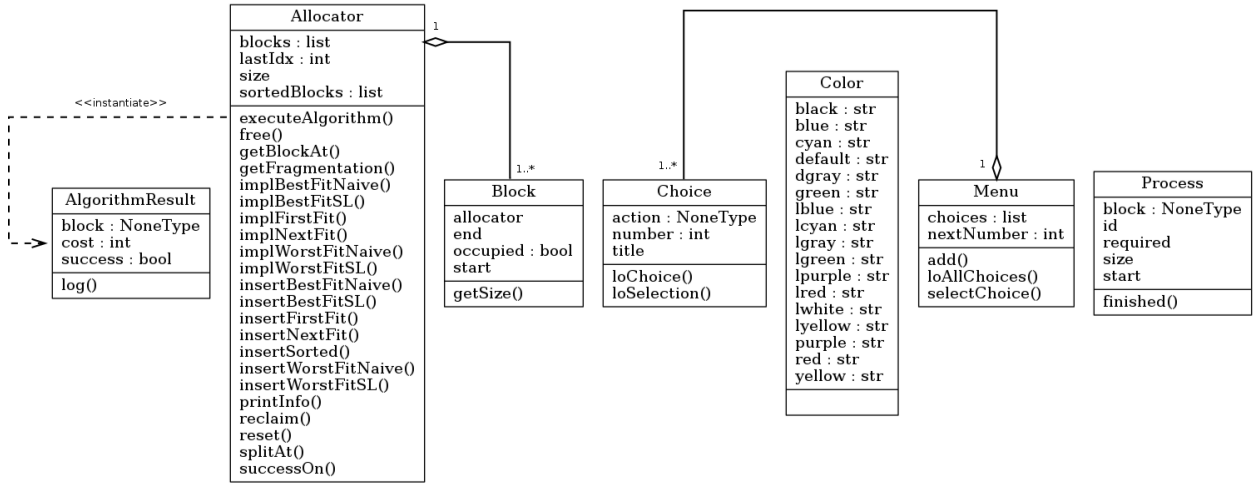
- **Modalità manuale**: gestita dalla funzione Python `mainManual` - questa modalità permette all'utente di interagire manualmente con l'**allocatore**, eseguendo allocazioni singole e visualizzando tramite un **grafico ASCII** il loro risultato.

- **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.

3. Diagramma UML

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.



4. 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.

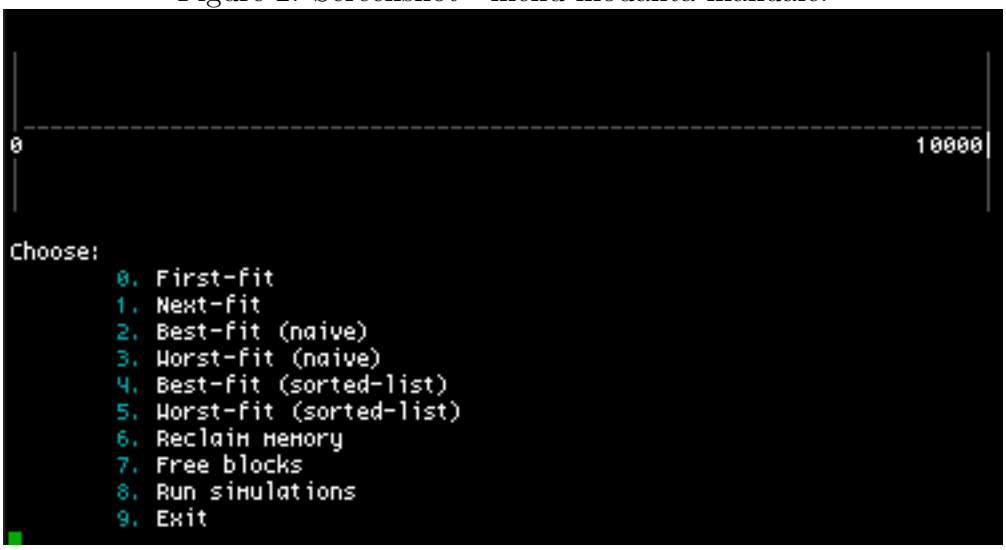
4..1 Modalità manuale

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

```
1 ./project.py
```

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

Figure 2: Screenshot - menù modalità manuale.



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

4..2 Modalità automatica

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

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

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