



SAPIENZA
UNIVERSITÀ DI ROMA

Scheduler dei processi di un sistema operativo, confronto ed analisi di varie politiche

Facoltà di Ingegneria dell'informazione, informatica e statistica
Corso di Laurea in Ingegneria informatica ed automatica

Candidato

Simone Trenta

Matricola 1724141

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2020/2021

Tesi discussa il Marzo 2021

di fronte a una commissione esaminatrice composta da:

Prof. ... (presidente)

Prof. ...

Prof. ...

Prof. ...

Prof. ...

Prof. ...

Prof. ...

Scheduler dei processi di un sistema operativo, confronto ed analisi di varie politiche

Tesi di Laurea. Sapienza – Università di Roma

© 2021 Simone Trenta. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: trenta.1724141@studenti.uniroma1.it

Sommario

Tramite un progetto di simulazione di uno scheduler di un sistema operativo, si testano ed analizzano le varie politiche di scheduling fra le più comuni, ossia first come first served (FCFS), round robin (RR), shortest job first (SJF) e shortest remaining job first (SRJF).

Indice

1	Introduzione	1
2	Lo scheduler in un sistema operativo	2
2.1	La sua utilità	2
2.2	Politiche di scheduling	2
2.2.1	First come first served	3
2.2.2	Round robin	3
2.2.3	Shortest job first	3
2.2.4	Shortest remaining job first	4
3	Sviluppo ed implementazione del simulatore	5
3.1	Il progetto	5
3.2	Libreria process.h	5
3.3	Libreria process_list.h	6
3.4	Libreria setting.h	6
3.5	Libreria stat.h	7
3.6	Libreria utility.h	7
3.7	Libreria macro.h	8
3.8	Libreria fcfs.h	8
3.9	Libreria rr.h	9
3.10	Libreria sjf.h	10
3.11	Libreria srjf.h	11
4	Esempi d'uso	13
4.1	Casi analizzati	13
4.2	Analisi casi in first come first served	14
4.3	Analisi casi in round robin	17
4.4	Analisi casi in shortest job first	19
4.5	Analisi casi in shortest remaining job first	22
5	Conclusioni	25
	Bibliografia	26

Capitolo 1

Introduzione

In un sistema operativo la gestione delle richieste di risorse da parte dei processi è demandata allo scheduler. Questa funzione di un sistema operativo è pressoché essenziale, senza la quale le risorse disponibili non potrebbero essere riservate e utilizzate in modo efficiente.

Nel **Capitolo 2** si descrive uno scheduler ed i vari algoritmi che verranno poi implementati nel simulatore.

Nel **Capitolo 3** viene fatta una panoramica su tutte le librerie del simulatore scritte. In particolare ci si concentra sulle librerie che permettono l'implementazione delle politiche di scheduling.

Nel **Capitolo 4** si analizzano i risultati dell'esecuzione del simulatore con vari casi di test, studiandoli per ogni singola politica.

Capitolo 2

Lo scheduler in un sistema operativo

2.1 La sua utilità

Un odierno sistema operativo è composto da molteplici parti, una di esse è lo scheduler dei processi. Questi ultimi sono parte costituente del sistema operativo, sono la parte esecutiva di esso. Per meglio capire, i processi richiedono delle risorse del calcolatore, che siano risorse unicamente della CPU oppure di risorse di I/O. Qualunque tipo di risorsa richiedano, spesso non è immediatamente disponibile, essendo limitate in un calcolatore. Per cui è necessaria la presenza di un elemento che permetta al dispositivo di gestire le proprie risorse, tale elemento è per l'appunto lo scheduler. Vedendo il problema da parte dei processi, problema che è la possibilità di usare una risorsa o meno, si crea la necessità di dare delle priorità per poterne usufruire. Andando quindi a dare origine a tali priorità si producono di conseguenza delle politiche per poterle mettere in atto.

2.2 Politiche di scheduling

Le politiche di scheduling sono degli algoritmi, che permettono quindi di riservare le risorse del calcolatore in un certo modo. Ognuna di esse ha, come ogni cosa, dei vantaggi e degli svantaggi. Tale prospettiva non permette di rendere efficiente una unica politica per ogni tipo di calcolatore, ma sarà necessario dotare ogni sistema operativo con l'algoritmo più adatto. Alcuni parametri che vengono presi in considerazione per valutare l'efficienza di un algoritmo sono:

- Utilizzo della CPU
- Throughput del sistema
- Tempo di attesa di un processo
- Turnaround di un processo
- tempo di risposta di un processo

Fra questi non tutti potranno essere ottimizzati allo stesso tempo, in particolare nei sistemi batch si massimizza throughput e minimizza il turnaround, mentre nei sistemi interattivi si minimizza il tempo medio di risposta ed il tempo di attesa. I parametri considerati nel progetto implementato sono il tempo di attesa e il tempo

di completamento di un processo. Quest'ultimo da considerarsi come il tempo totale che ogni processo ha impiegato per terminare, per cui partendo sempre dallo stesso valore per il tempo massimo iniziale, se il tempo di completamento sarà elevato vorrà dire che il processo ha impiegato molto per concludersi, rimanendo talvolta in attesa.

2.2.1 First come first served

L'algoritmo *first come first served*, forse, è il più semplice da capire, e da implementare. Come dice il nome stesso esso non ha una particolare complessità, semplicemente quando un processo arriva, se è disponibile la risorsa, quindi un core della cpu, esso viene messo in running. Altrimenti viene messo in una coda di waiting, dalla quale vengono presi i processi da mettere in running, prima di considerare i nuovi processi in arrivo. In questa politica può risultare che un processo debba attendere la terminazione di un altro prima di poter entrare in running. Per cui può risultare che dovrà attendere molto in stato di waiting, nascendo quindi il problema di attesa lunghe se le durate dei processi sono elevate.

2.2.2 Round robin

L'algoritmo *round robin* è una successiva perfezione del precedente fcfs (first come first served). Anche in questo caso è presente una coda dei processi che sono arrivati. Se è disponibile la risorsa richiesta dal processo in arrivo, gli viene riservata e tale processo può sfruttarla, altrimenti viene inserito nella coda di attesa. Un processo in running, però, non ha la risorsa riservata fino a quando lo necessita, ma in questo caso è presente un clock, che inizia a scorrere appena il processo entra in running. Terminato il clock, la risorsa viene liberata dal processo. Esso viene inserito nella coda dei processi in attesa, e da essa viene preso il primo processo per destinarli la risorsa appena liberatesi. Il problema di questo algoritmo è scegliere un valore ottimale per il clock, se troppo elevato si tenderà ad avere un approccio simile a fcfs, se non uguale, se troppo breve i processi non riusciranno a sfruttare la risorsa richiesta che dovranno liberarla.

2.2.3 Shortest job first

L'algoritmo *shortest job first* ordina i processi da mettere in running, ovviamente se c'è disponibilità di risorsa, in base a quanto tempo essi la richiedono, dal più breve al più lungo. Anche in questo algoritmo un processo entrato in running termina completamente il suo job prima di terminare. Se la risorsa non è disponibile, il processo viene inserito in una coda, il cui ordinamento è basato sulla durata del job. Inoltre tutti i processi da mettere in running vengono presi da questa coda a cui si aggiungo i processi che arrivano in un determinato istante. In questo algoritmo risulterà che i processi con la richiesta minore di risorsa saranno schedulati e termineranno per primi. Il tempo medio di attesa si riduce, però il problema principale di questa politica nella realtà è riuscire a predire, con sufficiente accuratezza, quanto un processo occuperà una risorsa. Nel caso del simulatore sviluppato ciò non è necessario, poiché ogni processo che arriva dichiara per quanto tempo occuperà la risorsa. Al contrario, realisticamente, si utilizza la seguente formula per predire l'utilizzo successivo della cpu:

$$\hat{b}_{t+1} = \alpha b_t + (1 - \alpha) \hat{b}_t \quad (2.1)$$

In cui si ha:

$$\hat{b}_{t+1} \quad (2.2)$$

è la predizione al tempo $t+1$;

$$\alpha \quad (2.3)$$

è il coefficiente di decadimento;

$$b_t \quad (2.4)$$

è la misura avvenuta al tempo t ;

$$\hat{b}_t \quad (2.5)$$

è la predizione al tempo t .

2.2.4 Shortest remaining job first

L'algoritmo *shortest remaining job first* è simile al precedente sjf. Però in questo caso ad ogni ciclo di clock della cpu, i processi in running vengono confrontati con tutti quelli in waiting e quelli appena arrivati. I processi, sempre in base alla disponibilità della risorsa, vengono messi in running, o ci rimangono, in ordine in base alla durata del job, dal più piccolo al più grande. Anche in questo caso il problema principale è riuscire a predire la durata della richiesta della risorsa, e si utilizza la formula (2.1).

Capitolo 3

Sviluppo ed implementazione del simulatore

3.1 Il progetto

Il progetto sviluppato permette di simulare uno scheduler con le relative politiche presentate nella [sezione 2.2](#). L'implementazione è stata fatta usando il linguaggio di programmazione C, utilizzando il paradigma di programmazione funzionale con strutture dati create ad hoc. Per lo scopo sono state scritte varie librerie apposite, a seguire la spiegazione di ogni libreria, mentre per consultarle si può andare alla repository [\[1\]](#) dove, inoltre, è presente tutto il codice del progetto.

3.2 Libreria `process.h`

Nella libreria `process.h` [\[2\]](#) viene implementato un tipo di dato per simulare un processo. Per poter fare ciò si è fatto uso di una struttura dati, con all'interno sia tipi di dati primitivi, che tipi di dato enumerati. La struttura creata, chiamata `ProcessType`, contiene al suo interno:

- un intero, chiamato `pid`, indicante il pid del processo simulato;
- un intero, chiamato `time_arrive`, indicante il tempo di arrivo;
- un intero, chiamato `duration`, indicante la durata del job;
- un tipo di dato enumerato, chiamato `ResourceType`, indicante che tipo di risorsa si richiede. Può assumere i valori 0, per chiedere risorse prettamente di CPU, oppure 1, indicante che oltre alla CPU ha una parte di interattività con l'utente, richiedendo risorse di tipo I/O;
- un tipo di dato enumerato, chiamato `StateType`, indicante lo stato in cui si trova il processo. Può assumere i valori, 0 per lo stato NOT_STATE, 1 per RUNNING, 2 per WAITING, 3 per READY, 4 per TERMINATED e 5 per BURST.

Lo stato NOT_STATE indica che il processo è stato creato, ma ancora non è arrivato. Lo stato RUNNING significa che il processo è in esecuzione, ha una risorsa riservata e sta procedendo. Lo stato WAITING vuol dire che il processo non ha trovato risorse disponibili, è in attesa che se ne liberino. Lo stato READY indica che il processo è appena arrivato e deve ancora essere schedato. Lo stato TERMINATED

significa che il processo ha terminato completamente la sua esecuzione ed è uscito dallo scheduler. Lo stato BURST, invece, vuol dire che il processo ha terminato una sua esecuzione, ma richiederà nuovamente delle risorse successivamente.

In tale libreria sono presenti funzioni per la gestione di tali processi. Come funzioni per crearli partendo dai dati passati come parametri, per stamparli a video o in un file, controllare in che stato siano o che risorsa hanno richiesto, cambiarne lo stato ed anche richiedere un nuovo burst (nuova risorsa). Quest'ultima funzione è particolarmente importante, in quanto permette al processo di richiedere una nuova risorsa, rimanendo così nel ciclo dello scheduler. Ciò è implementato andando a generare numeri pseudo-casuali tramite la funzione di libreria standard *rand*, opportunamente scalata su numeri adeguati, in base a ciò che si richiede. Per un nuovo tipo di risorsa si scalerà i numeri in un range di 2, risultando i numeri calcolati o 0 o 1. Per la durata del nuovo job, invece, si calcola come massimo il tempo medio voluto inizialmente, così che il job potrà durare da 0 al tempo medio. Per il nuovo tempo di arrivo è calcolata partendo dal tempo attuale e come massimo il tempo massimo iniziale voluto moltiplicato per due.

3.3 Libreria `process_list.h`

Nella libreria *process_list.h* [3] viene creata una lista concatenata di processi, con il tipo di dato spiegato nella [sezione 3.2](#). I suoi elementi sono strutture dati così composte:

- un puntatore ad una struttura dati *ProcessType*, in cui sono contenute tutte le informazioni del processo in lista;
- un puntatore all'elemento successivo.

Tale struttura dati prende il nome di *ProcessItem*. La struttura dati rappresentante la lista concatenata è costituita da 2 elementi:

- un intero, chiamato *size*, indicante la dimensione della lista, ossia quanti processi ci sono in essa;
- un puntatore ad un dato *ProcessItem*, il quale è il primo elemento della lista.

In questa libreria sono presenti funzioni per una lista partendo dai dati letti da un file, per stamparla a video o in un file, contare i numeri di processi che sono in un determinato stato, generare una nuova richiesta di burst per i processi che ne sono in condizione. Di queste funzioni una importante è quella che permette di inserire un nuovo elemento nella lista. Tale funzione lo inserisce in testa, permettendo quindi di avere un costo costante pari a $O(1)$, soluzione ottima per l'inserimento degli elementi in una lista.

Inoltre ci sono funzioni per la gestione delle statistiche, aggiornamento tempi di attesa e di completamento, che grazie al tipo di dato ed ad altre funzioni implementate nella libreria descritta nella [sezione 3.5](#), le tengono sempre aggiornate.

3.4 Libreria `setting.h`

La libreria *setting.h* [4] permette di creare un tipo di dato contenente le impostazioni iniziali su cui far girare il simulatore. Tale tipo di dato è una struttura chiamata *SettingType* contenente al suo interno:

- un intero, indicante il numero di core, chiamato *core*;
- un intero, indicante il numero di processi, chiamato *pid*;
- un intero, indicante il tempo medio del job dei processi, chiamato *avg_time*;
- un intero, indicante il tempo massimo iniziale di arrivo dei processi e durante il quale ogni processo chiederà nuovi burst, chiamato *max_time*.

In questa libreria è implementata la funzione grazie alla quale vengono letti i dati in lista da un file e viene creato un puntatore ad un tipo di dato *SettingType*. Inoltre è presente una funzione per stampare a video i dati contenuti in tale struttura.

3.5 Libreria stat.h

La libreria *stat.h* [5] permette di gestire le statistiche createsi durante la simulazione. Ciò avviene tramite una struttura che prende il nome di *StatisticsType*, la quale è composta dai seguenti dati:

- un intero, per indicare il numero di processi, chiamato *num_proc*;
- un intero, indicante il numero di core, chiamato *core*;
- un array di interi, di grandezza *num_proc*, per memorizzare il tempo di attesa di ogni singolo processo, chiamato *waiting_time*;
- un array di interi, di grandezza *num_proc*, per salvare in memoria il tempo in cui i processi sono rimasti nello scheduler, o anche il tempo totale di completamento di ogni singolo processo, chiamato *completing_time*;
- un float, che indica il tempo medio di attesa di tutti i processi, chiamato *medium_wait_time*;
- un float, indicante il tempo medio di completamento di tutti i processi, chiamato *medium_complete_time*.

Sono presenti funzioni, che oltre alla creazione della struttura dati, permettono di aggiornare le statistiche, calcolando la media dei tempi di attesa e completamento. Sono implementate funzioni per la stampa a video e per il salvataggio in un file, quest'ultima ci permetterà poi di avere dati finali grazie ai quali potremo confrontare i vari algoritmi di scheduling implementati.

3.6 Libreria utility.h

Con la libreria *utility.h* [6] vengono implementate funzioni per l'utilizzo comune in più politiche di scheduling. In particolare sono sviluppate funzioni per la gestione di un array di interi, il quale memorizzerà i pid dei vari processi, e sarà gestito secondo l'algoritmo specifico da implementare. Inoltre sono presenti funzioni per la determinazione della durata iniziale dei job dei processi, che viene richiesto abbiano una durata media desiderata.

3.7 Libreria macro.h

L'ultima libreria prima di passare alle librerie in cui vengono implementati gli algoritmi di scheduling è la libreria *macro.h* [7]. In questa libreria non sono presenti funzioni, ma vengono unicamente definite macro di utilizzo globale nel simulatore, come i file da cui leggere i dati, e dove scrivere le statistiche finali create. Sono presenti anche definizioni per richiamare le varie librerie fin'ora descritte e tutte le successive in cui ci sono le politiche di scheduling. Per finire ci sono anche delle variabili numeriche per la definizioni del numero complessivo di set utilizzati, del numero di cicli per cui il simulatore finale dovrà girare ed anche il quanto di tempo utilizzato in RR.

3.8 Libreria fcfs.h

Iniziamo con la libreria *fcfs.h* [8] a descrivere le varie implementazioni degli algoritmi di scheduling, riprendendo l'ordine in cui sono stati presentati nella [sezione 2.2](#). Per cui la prima politica sarà first come first served. Per meglio descrivere la sua implementazione faremo riferimento allo schema a stati finiti di figura [3.1]. Ogni

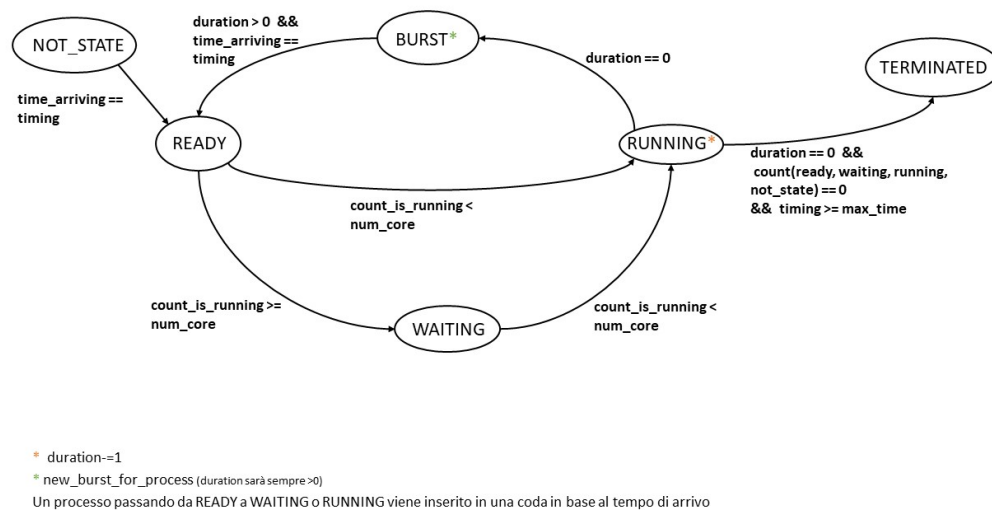


Figura 3.1. Schema a stati finiti FCFS

processo simulato (da ora in avanti per non appesantire la scrittura sarà denominato semplicemente processo, ma si intenderà sempre simulato) appena creato si trova nello stato NOT_STATE. Viene inserito in una lista con tutti i processi, viene creato un array contenente i pid dei processi che arriveranno. Il simulatore inizia il suo percorso, andando a procedere per ogni singolo clock del processore (anche questo simulato), chiamato *timing*. Ad ogni clock si controlla nella lista i processi che hanno tempo di arrivo uguale a *timing*, nel qual caso si cambia il loro stato in READY. Dopo di ciò, per tutti i processi in stato READY ed in stato di WAITING, quest'ultimi già in coda, e con precedenza a questi, se è disponibile la risorsa, quindi un core, il processo va in stato RUNNING. Si contano nuovamente i processi in RUNNING, se tale numero è minore del numero di core disponibili, si procede ad inserire il successivo processo da READY a RUNNING. Altrimenti, se non c'è più

disponibilità di risorse, tutti i processi vengono inseriti in una coda in base al tempo di arrivo e il loro stato cambia in WAITING. Ogni processo in RUNNING diminuisce di una unità la durata del suo job ad ogni ciclo di clock. Quando la durata è uguale a 0, il processo cambia stato passando in BURST, liberando anche una risorsa. In tale stato al processo vengono assegnati nuovi tempi di arrivo, nuova risorsa, nuova durata del job, grazie alla funzione descritta fine [sezione 3.2](#). Quando timing sarà uguale max_time, descritto nella [sezione 3.4](#), si passa ad un ciclo in cui i processi procedono come sopra descritto, ma quando il job sarà uguale a 0 non andranno in BURST, ma passeranno in TERMINATED, non richiedendo ulteriori risorse. Una volta che tutti i processi in lista sono TERMINATED il simulatore con politica FCFS termina. Ad ogni ciclo di clock, vengono sempre aggiornate le statistiche, per poi essere scritte in un file a terminazione della simulazione.

3.9 Libreria rr.h

La libreria *rr.h* [9] implementa l'algoritmo round robin. Prima di procedere all'analisi degli stati dei processi, in questa libreria viene implementato un ulteriore tipo di dato come struttura, per gestire i core ed il loro tempo ancora disponibile se utilizzati. Tale struttura dati contiene:

- un intero, indicante il pid del processo che sta utilizzando il core, chiamato *pid*;
- un intero, indicante il tempo rimanente per l'uso del core, chiamato *core_time*.

Prendendo in considerazione lo schema a stati finiti dell'immagine [3.2] analizziamo gli stati dei processi. Ogni singolo processo appena creato si trova in stato NOT_STATE

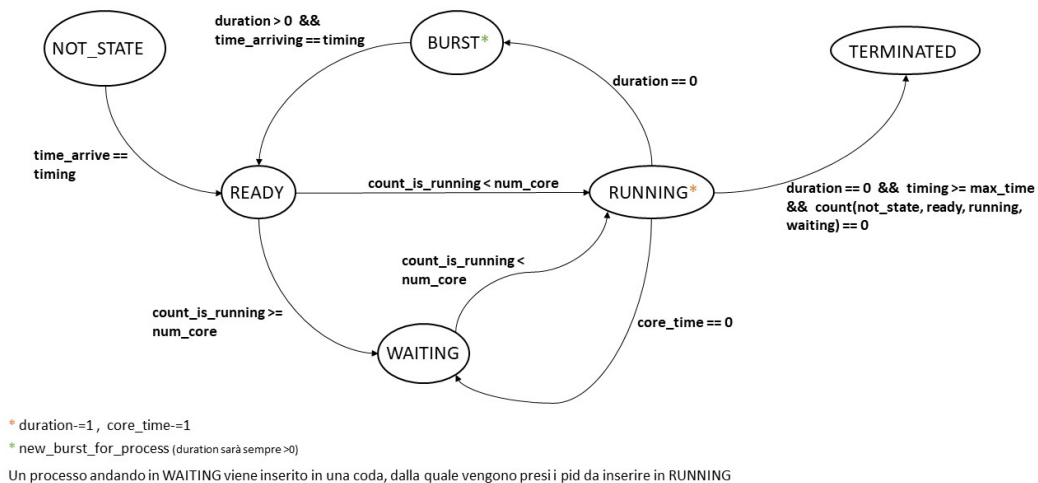


Figura 3.2. Schema a stati finiti RR

e viene inserito nella lista dei processi. Il simulatore inizia il suo primo ciclo, ad ogni clock cerca nella lista dei processi quelli per cui time_arrive è uguale a timing. Tali processi passano allo stato READY, essendo arrivati, e vengono inseriti in una

coda per poterli gestire in base al loro arrivo. Se è disponibile la risorsa, quindi il numero di processi in RUNNING minore del numero di core, si prende il primo processo dalla coda e passa in RUNNING, reinizializzando il quanto del core al tempo prestabilito. Si procede così fino a quando non terminano i core. Tutti i rimanenti processi nella coda passano nello stato di WAITING. Per tutti i processi in RUNNING viene ridotta di una unità la durata del job e del tempo rimanente del core. Quando il core_time è uguale a zero il processo passa nella coda dei processi in WAITING, e viene liberato un core, una risorsa. Se invece la durata del job si riduce a zero il processo passa allo stato BURST. In questo stato, come nella precedente politica, vengono calcolati nuovi tempi di arrivo, nuova durata del job e nuova richiesta di risorsa per il processo, come spiegato a fine [sezione 3.2](#). Quando il tempo trascorso nel primo ciclo del simulatore risulta maggiore di max_time, si entra nell'ultimo ciclo, per cui il procedimento è come sopra descritto, ma quando un processo termina il job passa allo stato TERMINATED. Quando tutti i processi sono nello stato TERMINATED il simulatore termina. Ad ogni ciclo vengono aggiornate le statistiche, per poi essere scritte in un file, il quale è presente nella libreria macro.h, descritta nella [sezione 3.7](#), a terminazione della simulazione.

3.10 Libreria sjf.h

Proseguendo con l'analisi delle varie librerie in cui si implementano vari algoritmi di scheduling, la successiva è la libreria *sjf.h* [10]. Questa libreria implementa la politica di scheduling shortest job first. Come per le precedenti librerie fcfs.h e rr.h, faremo riferimento allo schema a stati finiti per descrivere al meglio i passaggi fra stati in un processo di figura [3.3]. Come nei precedenti casi, ogni processo appena

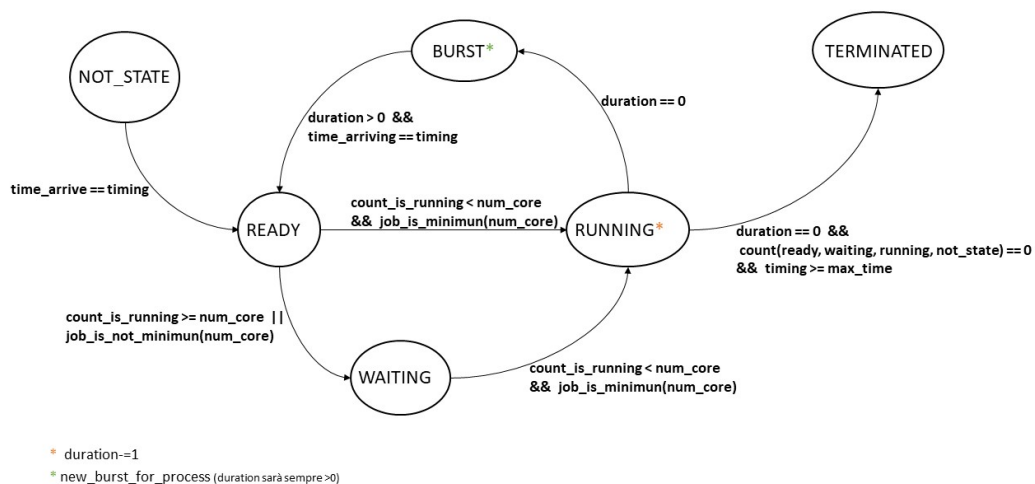


Figura 3.3. Schema a stati finiti SJF

creato viene inserito in una lista dei processi e si trova nello stato NOT_STATE. Si procede con il primo del simulatore, fino a quando timing non raggiungerà max_time. Ad ogni clock si controlla la lista, se un processo ha il tempo di arrivo uguale a timing passerà in stato READY. Per tutti i processi in READY e WAITING vengono

fatti i seguenti due controlli per poter passare in **RUNNING**, i quali devono risultare entrambi positivi:

- deve essere disponibile la risorsa richiesta, quindi in particolare almeno un core;
- la durata del job deve essere la minima fra tutti i processi in **READY**.

Come abbiamo detto nella [sezione 2.2.3](#), il problema di questo algoritmo è prevedere la durata del job di ogni singolo processo in arrivo, questo poiché nella realtà i processi non dichiarano per quanto tempo occuperanno una risorsa. Nel nostro caso, però, abbiamo fatto l'ipotesi che i processi lo dichiarino al loro arrivo. Così facendo si implementa facilmente una funzione che permette di verificare se un processo ha la durata minima fra tutti quelli in **READY**. Risultate positive entrambe le condizioni sopra descritte, il processo entra in **RUNNING**. Altrimenti entra in **WAITING**. Ogni processo in **RUNNING** diminuisce di una unità la durata rimanente del suo job ad ogni clock e rimane in questo stato fino alla terminazione del job. Quando la durata è uguale a zero il processo libera la risorsa fino ad allora occupa, ed entra in **BURST**. In questo stato, come per i precedenti algoritmi, viene calcolato un nuovo tempo di arrivo, nuova durata del job ed una nuova richiesta di risorsa per il processo, come spiegato a fine [sezione 3.2](#). Una volta terminato il primo ciclo del simulatore, ossia quando `timing` risulta uguale `max_time`, si entra nel secondo ciclo, ove il procedimento è lo stesso di quello sopra descritto. Unica cosa che cambia è che ogni processo terminato il proprio job non va in **BURST**, ma bensì in **TERMINATED**, non richiedendo ulteriori risorse. Quando tutti i processi saranno in **TERMINATED** il simulatore termina la sua esecuzione. Ad ogni ciclo vengono inoltre aggiornate le statistiche per poi essere scritte in un file a termine dell'esecuzione.

3.11 Libreria *srjf.h*

L'ultimo algoritmo implementato in questo simulatore è nella libreria *srjf.h* [\[11\]](#). Questa politica di scheduling è un miglioramento della precedente *sjf*, e come in essa il principale problema è riuscire a predire la durata del job di un processo in arrivo, ma come già detto, abbiamo fatto l'ipotesi che ogni processo in arrivo dichiari per quanto tempo richiede la risorsa. Anche in questo caso andiamo a descrivere al meglio tale politica con lo schema a stati a finiti nell'immagine [\[3.4\]](#). Come per gli altri algoritmi, ogni processo appena creato viene inserito in una lista con tutti i processi, ed è nello stato **NOT_STATE**. Dopo di ciò inizia il primo ciclo di scheduling, ed ad ogni clock se il tempo di arrivo del processo è uguale a `timing`, il processo entra in **READY**. Da qui, almeno teoricamente, la soluzione rispetto al *shortest job first* è più semplice, in quanto unica condizione che deve essere verificata è che la durata di un determinato processo sia la minore fra tutti quelli in **READY**, **WAITING** e **RUNNING**. Ciò avviene ad ogni clock, per un numero di processi uguale o minore al numero di core. Per cui ci saranno in **RUNNING** sempre i processi che risulta abbiano la durata del job minima fra tutti quelli negli stati **READY**, **WAITING** e **RUNNING**. Tutti i processi che non andranno in **RUNNING**, passeranno allo stato **WAITING**. Per gli altri processi in **RUNNING** si riduce di una unità la durata rimanente del job. Quando un processo termina il proprio lavoro entra nello stato **BURST**, dove richiederà delle nuove risorse, con una nuova durata ed un nuovo tempo di arrivo, come spiegato a fine [sezione 3.2](#). Una volta che il ciclo iniziale è terminato, ossia `timing` è maggiore o uguale a `max_time`, si entra nel ciclo finale. In esso i processi che terminano il job non entrano in **BURST**, ma vanno nello stato

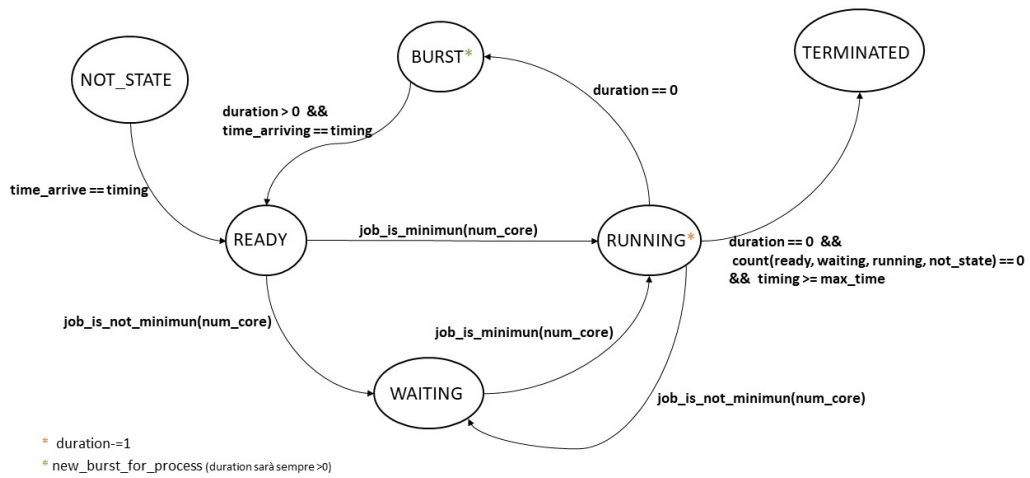


Figura 3.4. Schema a stati finiti SRJF

TERMINATED, ove non richiedono ulteriori risorse, ma semplice hanno terminato la propria esecuzione. Quando tutti i processi risulteranno in TERMINATED il simulatore termina. Durante ogni ciclo vengono sempre aggiornate le statistiche, che a fine simulazione vengono scritte in un file.

Capitolo 4

Esempi d'uso

4.1 Casi analizzati

Il simulatore implementato ha come corpo principale nel main un ciclo, nel quale vengono fatte partire una alla volta tutte le politiche sopra analizzate ed implementate. Successivamente i dati raccolti vengono cumulati, mantenendone la comprensibilità in base all'origine dei casi da cui sono scaturiti. Quest'ultimi sono ben 24 casi a politica, e per un'analisi più accurata ogni singolo caso è ripetuto 10 volte, per un totale di 240 situazioni a politica, per un complessivo di 960. Le chiamate alle funzioni di scheduling sono state effettuate con i seguenti casi:

- 1 core, 10 processi, tempo medio del job 10;
- 2 core, 10 processi, tempo medio del job 10;
- 4 core, 10 processi, tempo medio del job 10;
- 8 core, 10 processi, tempo medio del job 10;
- 16 core, 10 processi, tempo medio del job 10;
- 32 core, 10 processi, tempo medio del job 10;
- 1 core, 20 processi, tempo medio del job 10;
- 2 core, 20 processi, tempo medio del job 10;
- 4 core, 20 processi, tempo medio del job 10;
- 8 core, 20 processi, tempo medio del job 10;
- 16 core, 20 processi, tempo medio del job 10;
- 32 core, 20 processi, tempo medio del job 10;
- 1 core, 10 processi, tempo medio del job 20;
- 2 core, 10 processi, tempo medio del job 20;
- 4 core, 10 processi, tempo medio del job 20;
- 8 core, 10 processi, tempo medio del job 20;
- 16 core, 10 processi, tempo medio del job 20;

- 32 core, 10 processi, tempo medio del job 20;
- 1 core, 20 processi, tempo medio del job 20;
- 2 core, 20 processi, tempo medio del job 20;
- 4 core, 20 processi, tempo medio del job 20;
- 8 core, 20 processi, tempo medio del job 20;
- 16 core, 20 processi, tempo medio del job 20;
- 32 core, 20 processi, tempo medio del job 20.

Ora andremo ad analizzare i risultati ottenuti per ogni singola politica, verificando anche così successivamente nelle conclusioni finali i migliori usi per l'implementazione di ognuna. Premetto una breve descrizione dei grafici presenti successivamente. In essi sono presenti delle colonne, quelle arancioni rappresentano il numero dei processi, quelle blu il numero di core. Inoltre ci sono 2 curve, quella gialla rappresenta il tempo medio di completamento dei processi, mentre quella grigia, la più interessante, il tempo medio di attesa di ogni singolo processo. Essendo il tempo di completamento qui considerato come il tempo da quando un processo arriva allo scheduler a quando termina, questo fattore può essere considerato come un coefficiente di velocità dello scheduler; ciò significa che per un valore basso si avrà che saranno stati terminati più processi in un uguale arco di tempo ma con un corrispettivo valore di tempo medio di completamento più elevato.

4.2 Analisi casi in first come first served

Il primo metodo di scheduling in cui si sono andati ad acquisire i dati dei casi sopracitati è il *first come first served*. Ripetendo velocemente questo approccio, i processi vengono schedulati e portati ad esecuzione in base al loro tempo di arrivo, quello che arriva prima, viene schedulato e terminato per primo. Possiamo vedere grazie al grafico [4.1] i primi 6 casi, ossia da 1 a 32 core per 10 processi ed un tempo medio del job di 10. Possiamo notare subito dal grafico [4.1] l'andamento dei processi. Si vede come la saturazione dei core e delle relative risorse è subito presente, ma si attenua significativamente già quando si hanno 8 core, portando quasi a 0 il tempo di attesa. Con più core il tempo di attesa è praticamente nullo. Inoltre notiamo come anche il tempo medio di completamento si stabilizza dagli 8 core in poi, rimanendo stabile. Successivamente analizziamo il caso per cui abbiamo sempre da 1 core a 32 core per 10 processi, ma con un tempo medio del job di 20, ben rappresentato nel grafico [4.2]. In quest'ultimo notiamo come le 2 curve si appiattiscono molto velocemente. Infatti, nonostante si parte con valori relativamente elevati per entrambi, tempo di attesa e completamento, già con 2 core è presente un calo sostanziale. Per il completamento si ha una riduzione del circa 20%, mentre per l'attesa del circa 73%. Per entrambi si ha un assestamento, come nel caso precedente, a partire dagli 8 core, da cui il tempo di attesa è sostanzialmente nullo. Successivamente analizziamo il set di test per cui abbiamo da 1 core ai 32 core, 20 processi e un tempo medio a processo del job di 10. I risultati di questi casi sono presenti nel grafico [4.3]. Qui si evince ancor di più la rapida decrescita delle curve di nostro interesse all'aumentare dei core disponibili. Anche in questo caso la decrescita maggiore si ha da 1 core a 2 core, dove abbiamo una riduzione del 20% circa per il tempo di completamento, e del 68% circa per il tempo di attesa. In questo caso il

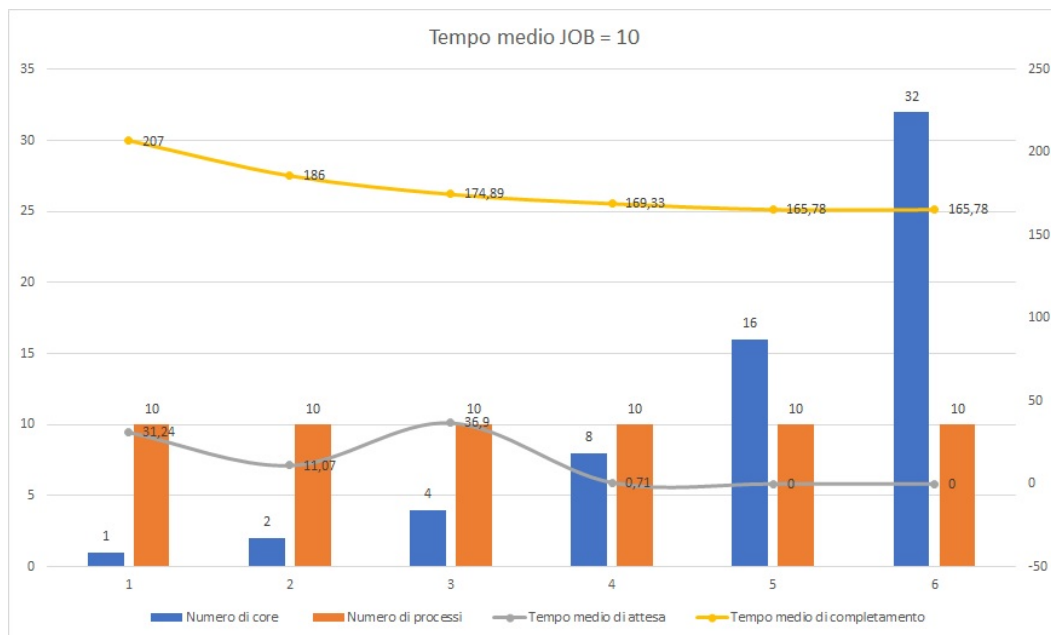


Figura 4.1. Grafico processi10 job10 fcfs

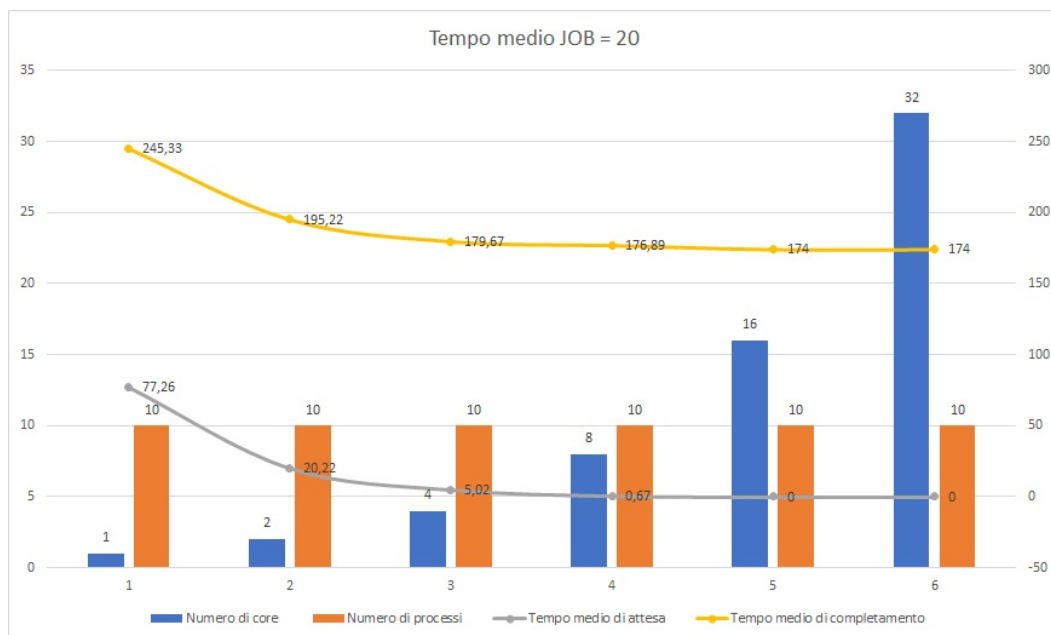


Figura 4.2. Grafico processi10 job20 fcfs

tempo di completamento rimane stabile dagli 8 core in poi, mentre quello di attesa si riduce a zero dai 16 core. Ciò ci fa intuire che in questi casi lo scheduler risulta saturo, anche per un numero maggiore di core disponibili rispetto ai precedenti casi. Come ultimo set di casi abbiamo quello da 1 a 32 core disponibili, 20 processi e un tempo per ogni singolo processo del job di 20. Il grafico [4.4] ne racchiude i relativi risultati. Anche in quest'ultimo grafico si desume che la decrescita delle due curve si ha nei primi aumenti dei core disponibili, ma rispetto ai casi precedenti si ha

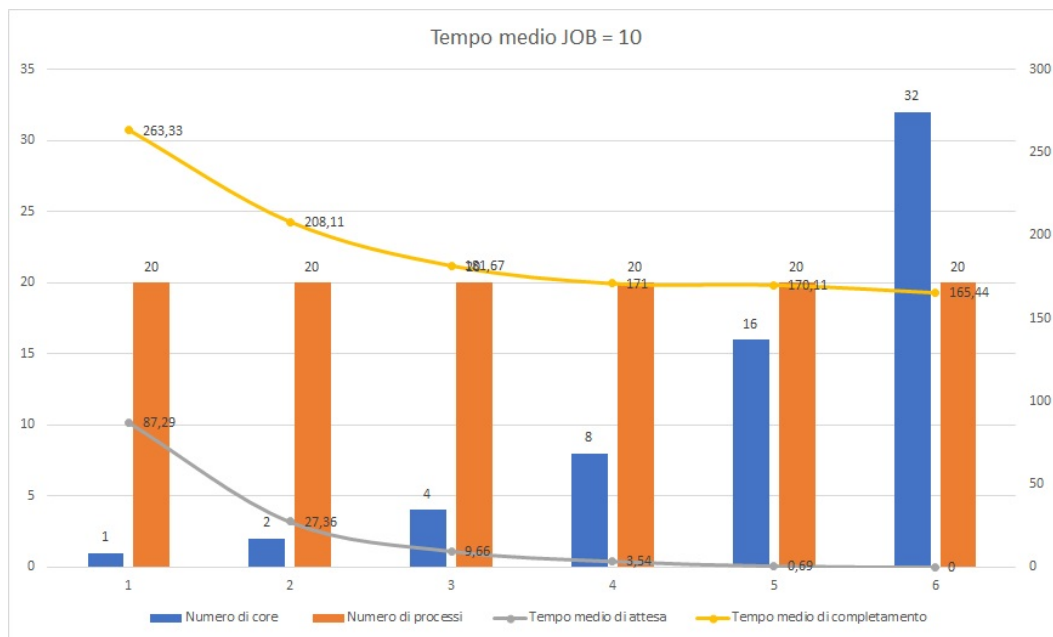


Figura 4.3. Grafico processi20 job10 fcfs

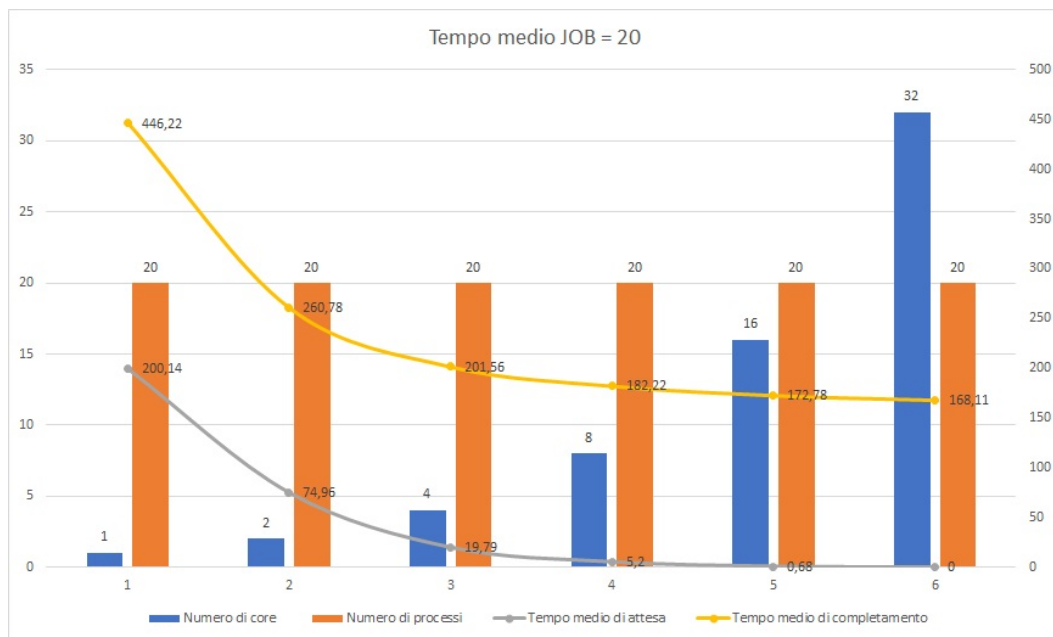


Figura 4.4. Grafico processi20 job20 fcfs

una riduzione importante anche al passaggio dai 2 ai 4 core. Tuttavia la decrescita maggiore si ha sempre da 1 core a 2 core, con una riduzione del 41% circa per il tempo di completamento, e del 62% circa per il tempo di attesa. Mentre al passaggio complessivo da 1 core a 4 core disponibili, si ha una riduzione del 54% circa per il completamento, e del 90% per l'attesa. La curva gialla rimane stabile a partire dai 16 core, mentre quella blu sostanzialmente si azzerava dai 16 core.

4.3 Analisi casi in round robin

Analizziamo ora la politica *round robin*, la quale, in breve, schedula i processi in ordine di arrivo, ma ad ognuno in esecuzione concede un quanto di tempo, scaduto il quale viene inserito in coda ai processi in attesa. Cominciamo tale indagine con i primi 6 casi, ossia da 1 a 32 core per 10 processi ed un tempo medio del job di 10. I risultati li possiamo vedere nel grafico a figura [4.5]. Da esso possiamo

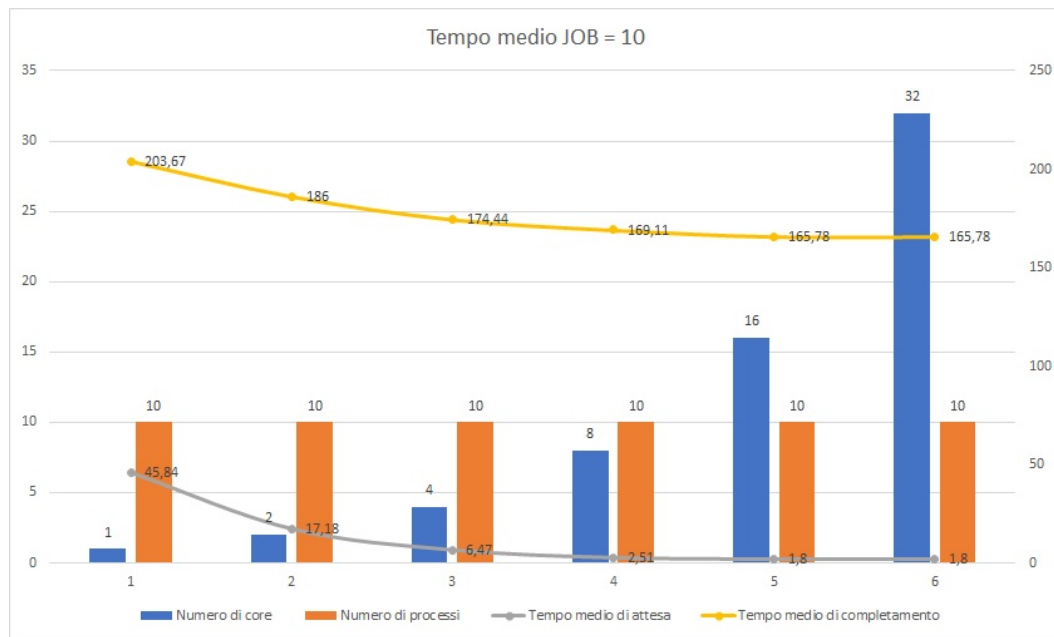


Figura 4.5. Grafico processi10 job10 rr

notare come la decrescita del tempo di attesa sia del circa 62% passando da un core a 2 core, ed un decremento simile, in percentuale, passando da 2 a 4 core. Vediamo, però, come il tempo di attesa non si azzerava mai, anche se comunque rimane di un valore relativamente piccolo da 16 core in poi. Mentre per il tempo medio di completamento vediamo che non c'è una diminuzione drastica, ma costante e relativamente minima, infatti passando da 1 a 4 core abbiamo una decrescita del 14% circa. Anche il tempo di completamento rimane costante da 16 core in poi. Successivamente confrontiamo i risultati per i test in cui sono presenti sempre 10 processi, da 1 a 32 core, ma il job medio richiesto è di 20. Il grafico con tali risultati è a figura [4.6]. Qui notiamo subito la forte diminuzione del tempo di attesa. Infatti passando da uno a due core si ha un tempo medio di attesa del -67% circa, ed un'ulteriore decrescita passando da 2 a 4 core, sempre del 67% circa. Mentre dagli 8 core rimane stabile, ma mai portandosi vicino allo 0, piuttosto rimane intorno a 5. Invece per il tempo medio di completamento si ha un'unica sostanziale diminuzione, passando da 1 a 2 core, corrispondente a circa il 18%. Mentre successivamente, oltre ad minima variazione del -6% circa da 2 a 4 core, non si hanno notevoli cambiamenti, e la curva rimane stabile intorno ad un valore compreso fra 180 e 174. Passando al set successivo di test, ossia per 20 processi con job medio di 10 e da 1 a 32 core, abbiamo nella figura [4.7] i relativi risultati. In questi casi notiamo la forte decrescita per il tempo di attesa medio, del 64% circa, nel primo aumento del numero di core disponibili. Mentre si ha un decremento del 85% circa passando da 1 a 4 core. Inoltre vediamo come la diminuzione sia presente in tutti i passaggi di aumento del numero

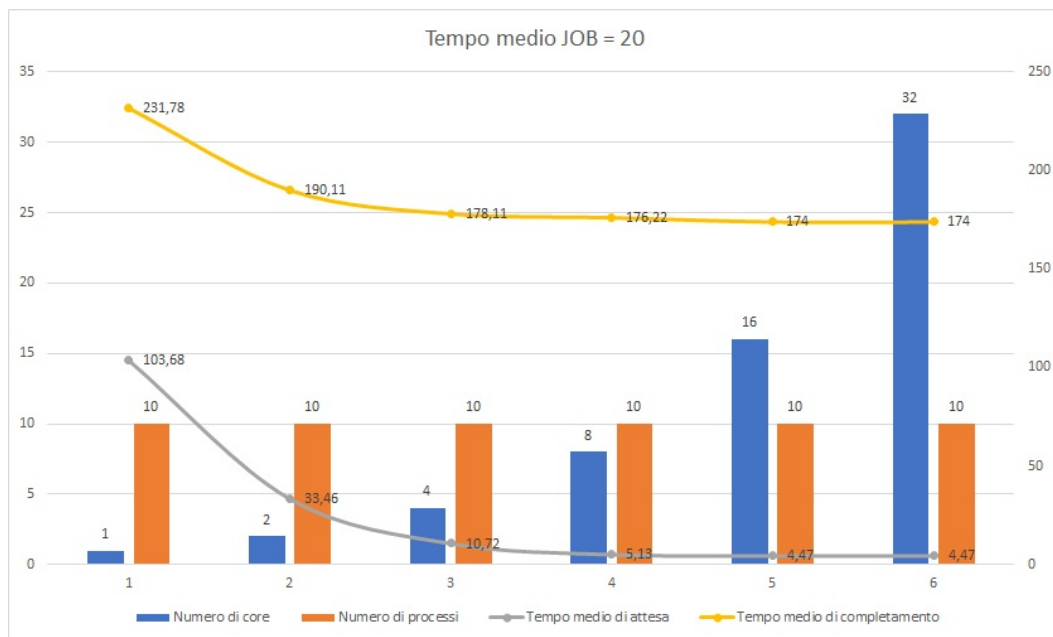


Figura 4.6. Grafico processi10 job20 rr

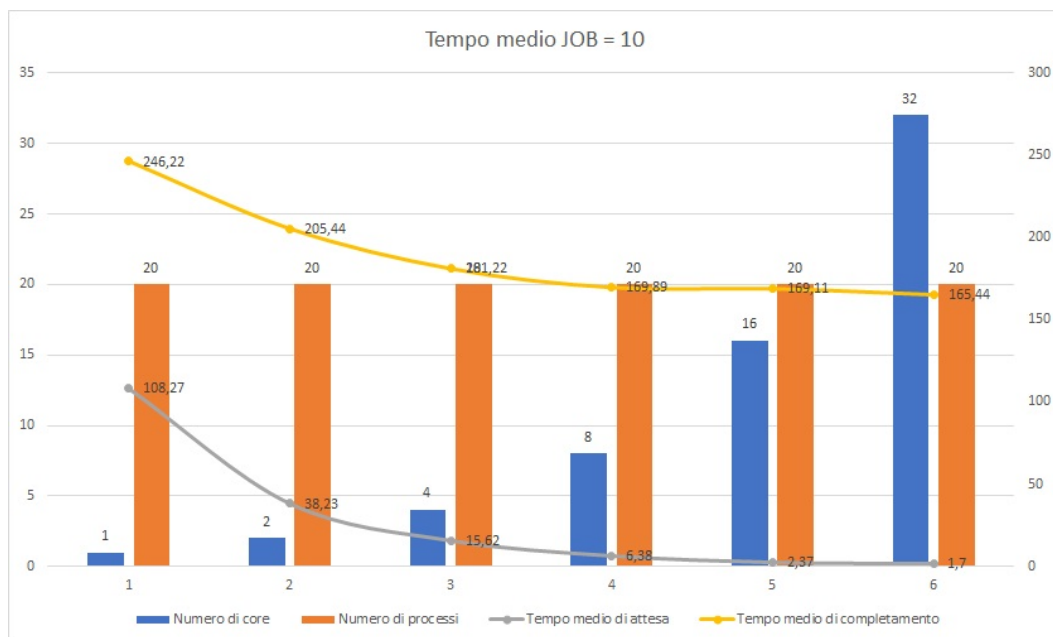


Figura 4.7. Grafico processi20 job10 rr

di core, terminando con un valore di 1,7 avendo 32 core disponibili. Per il tempo di completamento invece abbiamo una diminuzione del 16% circa passando da 1 a 2 core, e del 11% passando da 2 a 4. Anche successivamente, con l'aumentare del numero di core sono presenti dei decrementi nel tempo di completamento, ma non rilevanti come nei primi 2 casi. Nonostante ciò però abbiamo che la curva continua a deflettersi continuamente, arrivando per 32 core disponibili ad un valore di 165,44. Come ultima analisi per il round robin abbiamo i casi da 1 a 32 core, 20 processi

e job medio 20, con i risultati nel grafico a figura [4.8]. In questi casi vediamo

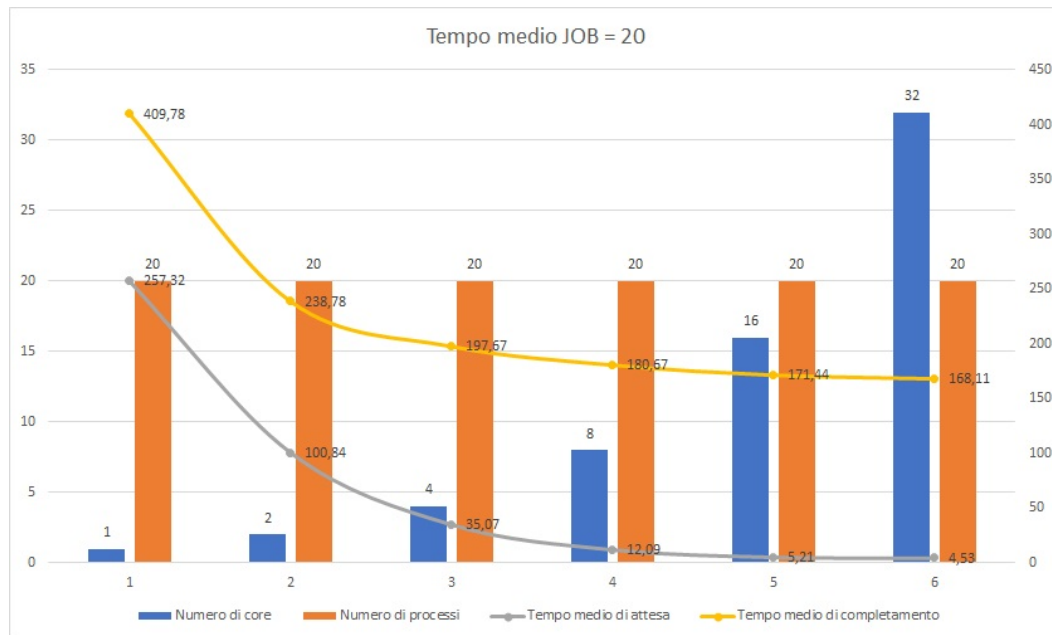


Figura 4.8. Grafico processi20 job20 rr

come con un solo core disponibile il tempo di attesa sia molto elevato, valore che diminuisce del 60% circa raddoppiando i core, riduzione presente anche passando da 2 a 4 core del 65% circa, e che rimane presente anche passando a 8 core. Mentre per 16 e 32 core il tempo di attesa rimane pressoché costante, attestandosi intorno al valore di 5. Passando invece alla curva del tempo di completamento vediamo come anche qui sia presente una notevole riduzione passando da 1 a 2 core, del 41% circa. Mentre successivamente, per gli ulteriori aumenti del numero di core la diminuzione sia costante attestandosi ad un -10% circa fino ai 16 core. Dai 16 ai 32 core invece il valore rimane stabile intorno a 170, con una leggera decrescita.

4.4 Analisi casi in shortest job first

Di seguito andiamo ad analizzare la politica *shortest job first*, la quale, in breve, schedula e manda in esecuzione i processi in base al tempo della risorsa che richiedono, dal più breve al più lungo, facendo terminare prima quelli già in esecuzione. Partiamo subito con il confronto dei risultati per i primi 6 casi, ossia per 10 processi, con tempo medio del job di 10 e numero di core disponibili da 1 a 32. Nella figura [4.9] ne troviamo i risultati. Da questo grafico vediamo immediatamente che il tempo di attesa, curva grigia, parte da un valore non troppo elevato e già nel primo passaggio si riduce del 65% circa. Mentre passando da 2 a 4 core si ha una riduzione pressoché totale, portando il tempo di attesa praticamente a zero, e rimanendo tale anche successivamente. Passando ad osservare la curva gialla, rappresentante il tempo medio di completamento, notiamo anche qui un calo passando da 1 a 2 core, del 12% circa. Successivamente non si hanno riduzioni consistenti, ed il tempo di completamento rimane stabile a partire dagli 8 core disponibili. I prossimi test che andiamo a visualizzare ed analizzare comprendono 10 processi, tempo medio del job richiesto di 20 e numero di core disponibili da 1 a 32. I risultati sono presenti nel

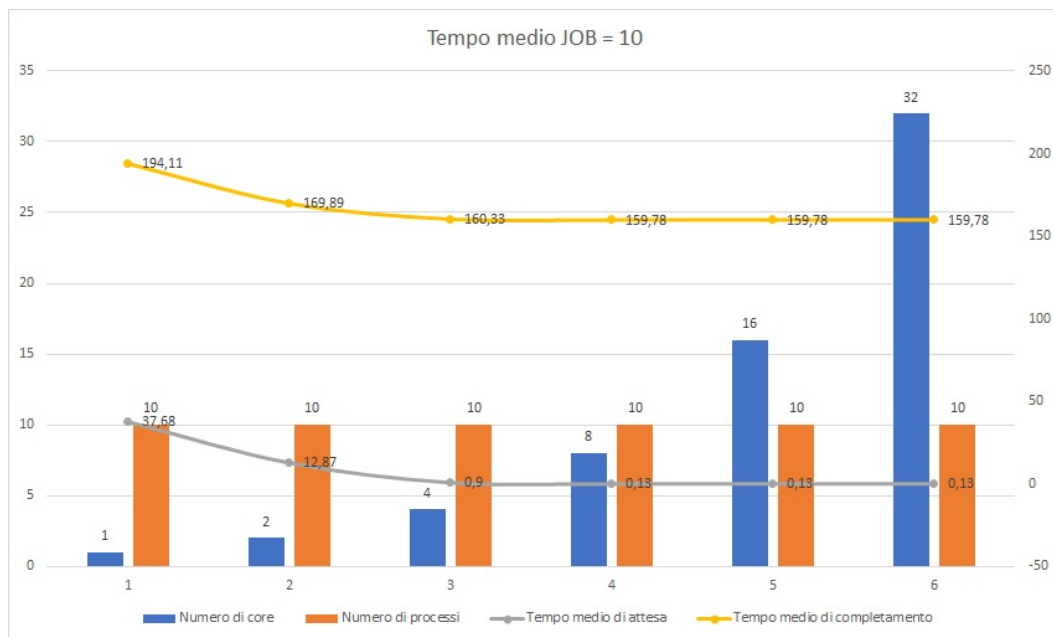


Figura 4.9. Grafico processi10 job10 sjf

grafico a figura [4.10]. Osservando il tempo di attesa vediamo come la riduzione sia

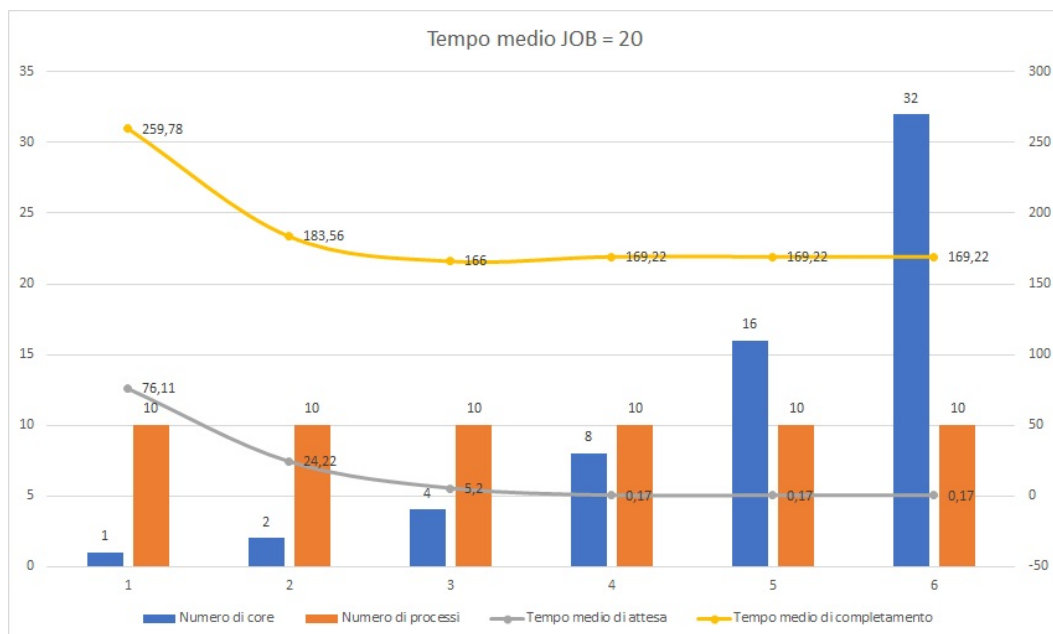


Figura 4.10. Grafico processi10 job20 sjf

presente fino ai 4 core, dopo il tempo di attesa si attesta e rimane stabile vicino a 0. Nel passaggio da 1 a 2 core si ha una diminuzione del 68% circa, mentre da 2 a 4 core del 78%, e successivamente si azzerà. Per il tempo di completamento abbiamo un comportamento analogo. Da 1 a 2 core si ha un decremento del 29% mentre da 2 a 4 core del 9%. Dai 4 core disponibili in poi la curva gialla rimane stabile.

Passiamo ai risultati per i test in cui ci sono 20 processi, tempo medio del job di 10 e da 1 a 32 core disponibili. Nella figura [4.11] è presente un grafico che ne racchiude i risultati. Anche in questi casi notiamo l'abbassamento della curva grigia, il tempo

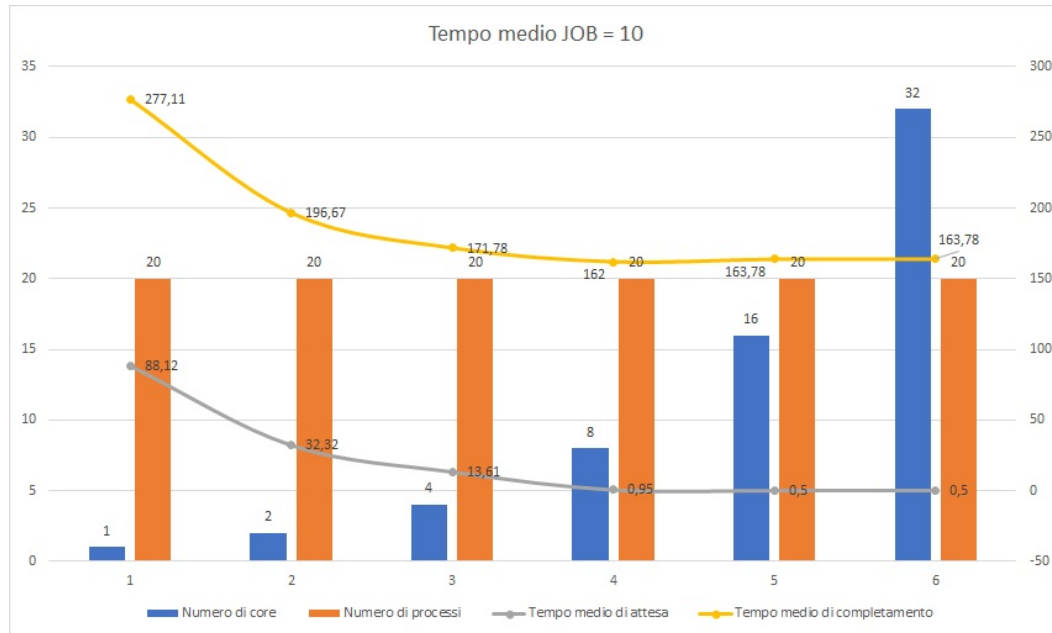


Figura 4.11. Grafico processi20 job10 sjf

medio di attesa, fino a quando si hanno 8 core, dai quali in poi rimane costante, e praticamente a 0. In particolare abbiamo una riduzione del 63% circa passando da 1 a 2 core, del 57% circa da 2 a 4 core ed un azzeramento passando agli 8 core disponibili. Per il tempo di completamento i risultati sono analoghi. Abbiamo delle riduzioni più o meno costanti fino ad avere 8 core disponibili, da cui in poi la curva gialla rimane stabile. Passando da 1 a 2 core si ha un cambiamento del -29% circa, da 2 a 4 del -12% circa, ed infine da 4 a 8 core di un -5% appena. Potremmo anche supporre che da 4 core in poi, essendo la riduzione da 4 a 8 minima, che il tempo di completamento rimane stabile. Gli ultimi risultati riguardano i test per 20 processi, con tempo medio del job richiesto di 20 e da 1 a 32 core disponibili, riassunti nel grafico a figura [4.12]. Partiamo con l'analisi del tempo di attesa, curva grigia, e notiamo come il calo sia costante e l'attesa si azzeri unicamente quando si hanno 32 core disponibili. Nei casi specifici abbiamo una riduzione del 54% circa da 1 a 2 core, del 65% circa da 2 a 4 core, del 77% circa da 4 a 8 core e di un'ulteriore 70% circa passando da 8 a 16 core. Passando da 16 a 32 core il tempo di attesa si azzerava. Osservando invece la curva gialla, rappresentante il tempo di completamento, abbiamo anche qui un calo fino agli 8 core, partendo da un valore elevato. In particolare abbiamo un calo del 41% circa passando da 1 a 2 core, del 28% circa da 2 a 4 core, del 11% circa da 4 a 8 core, mentre successivamente rimane stabile. C'è da notare come però passando dai 16 ai 32 core il tempo di completamento aumenta, anche se solo di poco (6% circa), dovuto ad una maggiore disponibilità di risorse.

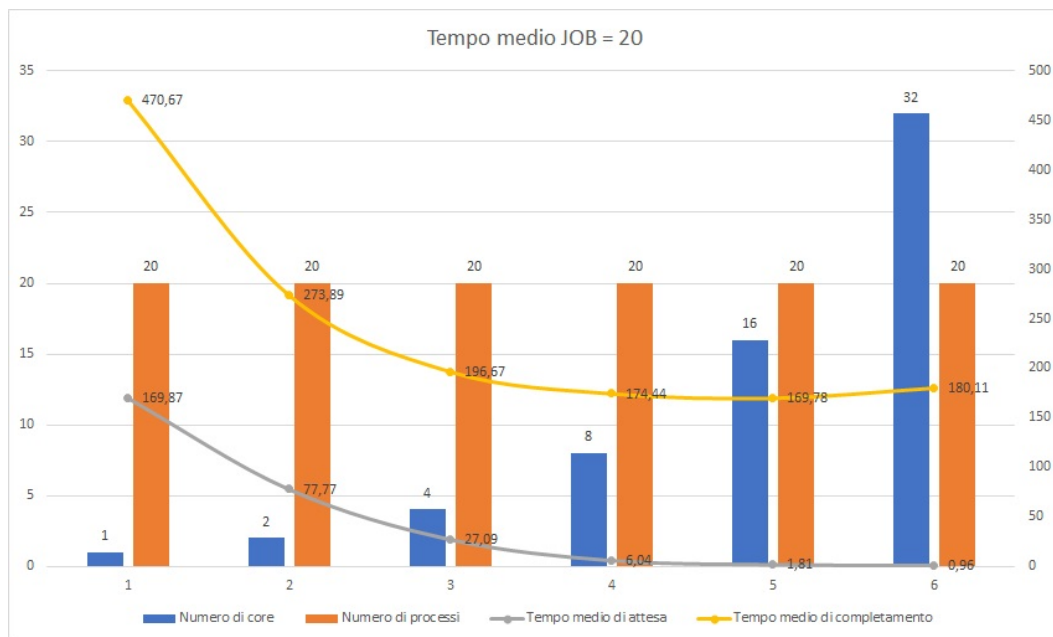


Figura 4.12. Grafico processi20 job20 sjf

4.5 Analisi casi in shortest remaining job first

Gli ultimi risultati che andiamo ad analizzare sono quelli derivati dalla politica *shortest remaining job first*. In essa i processi vengono schedulati come la precedente politica shortest job first, ma a differenza di essa, se arriva un processo che richiede meno tempo rispetto ad uno che è già in esecuzione, quest'ultimo viene messo nella coda dei processi arrivati, per far posto al processo con job richiesto minore fra tutti quelli arrivati fino a quel momento. I primi risultati che abbiamo sono quelli dovuti al set per cui si hanno 10 processi, tempo del job di 10 e da 1 a 32 core disponibili. Tali risultati sono raffigurati nel grafico a figura [4.13]. Possiamo subito notare delle sostanziali differenze rispetto ai casi precedenti. Infatti in questo caso non sono presenti delle diminuzioni apprezzabili nel tempo di completamento. La decrescita complessiva da 1 a 32 core è del 9% circa. Mentre nel tempo di attesa è presente un calo, del 50% circa da 1 a 2 core e del 20% passando da 2 a 4 core. Successivamente rimane stabile intorno ad un valore di 6. A seguire confrontiamo i risultati per i test dovuti a 10 processi, tempo medio del job di 20 e da 1 a 32 core disponibili, presenti nel grafico nella figura [4.14]. In questi casi vediamo come sia presente un calo sia del tempo di attesa che di completamento. Del primo, curva grigia, abbiamo un calo fino ai 4 core, dopo rimane stabile. In particolare abbiamo una riduzione del 41% circa passando da 1 a 2 core, e del 20% circa passando da 2 a 4 core. Per il tempo di completamento, invece, la riduzione significativa è fino ai 2 core, successivamente è presente ma non sufficientemente apprezzabile. Da 1 a 2 core abbiamo un decremento del 10% circa. Il terzo set di casi da esaminare è per 20 processi, tempo medio del job di 10 e numero di core disponibili da 1 a 32. Nella figura [4.15] possiamo vedere il grafico che ne raffigura i risultati. Qui abbiamo un calo, per il tempo di attesa, fino ai 4 core, e successivamente la curva grigia, che ne rappresenta l'andamento, rimane stabile. In particolare passando da 1 a 2 core disponibili abbiamo una decrescita del 56% circa, mentre da 2 a 4 core di un 17% circa. Per quanto riguarda il tempo di completamento il decremento è

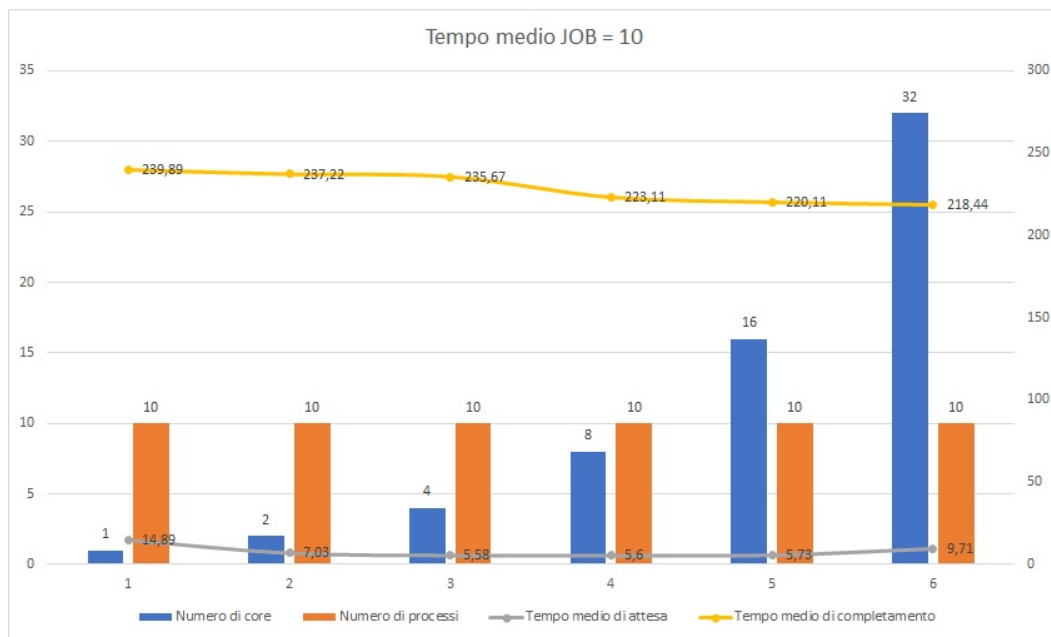


Figura 4.13. Grafico processi10 job10 srjf

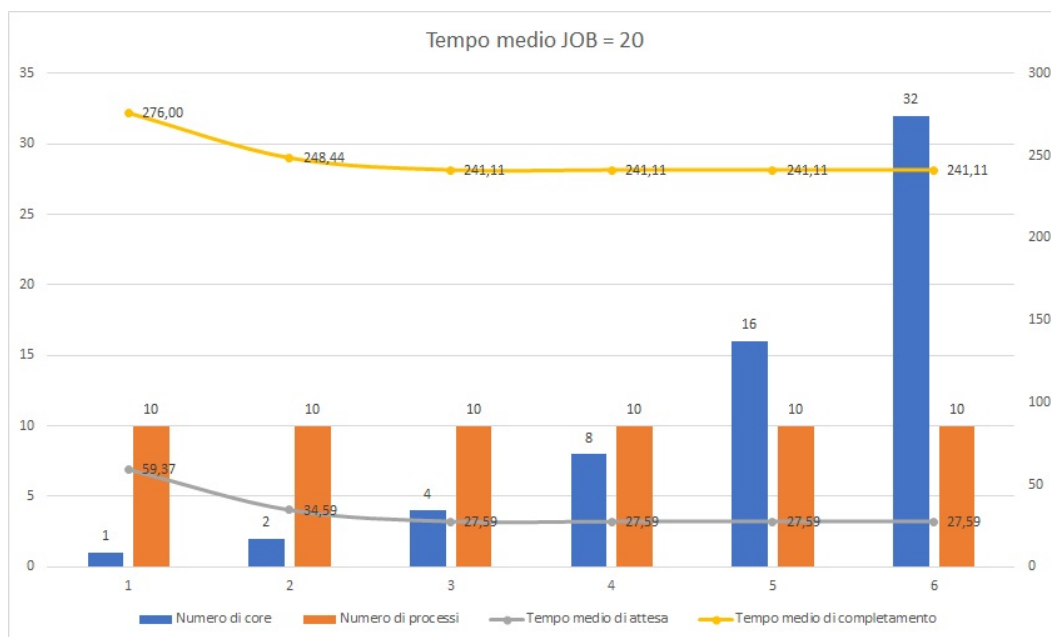


Figura 4.14. Grafico processi10 job20 srjf

presente anche qui fino ai 4 core per poi stabilizzarsi. Si ha un calo del 11% circa passando da 1 a 2 core e del 5% circa da 2 a 4 core. Gli ultimi casi che andiamo ad osservare sono per 20 processi, tempo medio richiesto per il job di 20 e numero di core disponibili da 1 a 32. I risultati di questi casi li possiamo vedere rappresentati nel grafico a figura [4.16]. Anche in questi casi abbiamo dei cali in entrambe le curve, per il tempo di attesa e di completamento, fino ai 4 core, punto da cui si assestano e rimangono stabili. Nel dettaglio abbiamo per il tempo di attesa un decremento del

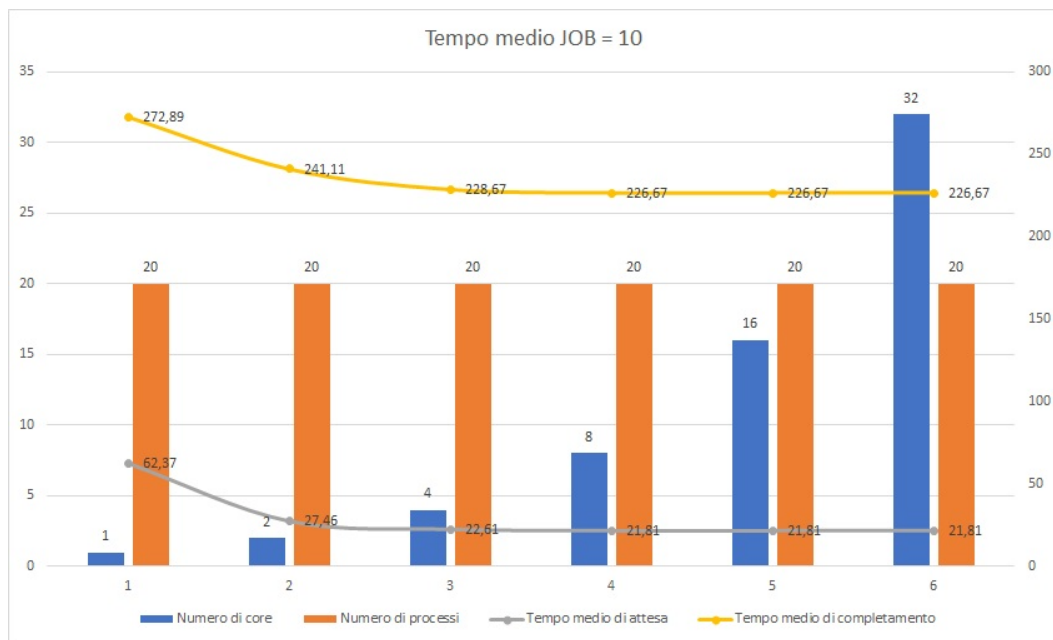


Figura 4.15. Grafico processi20 job10 srjf

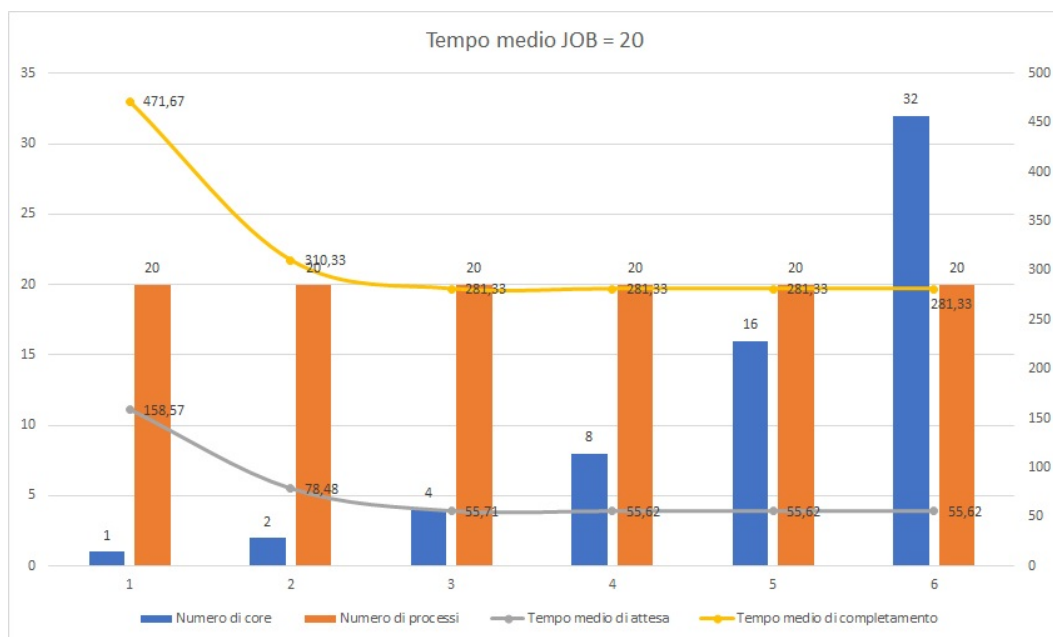


Figura 4.16. Grafico processi20 job20 srjf

50% circa passando da 1 a 2 core e del 29% circa da 2 a 4 core. Da notare come la relativa curva sia partita da un valore elevato. Per il tempo di completamento abbiamo un valore di partenza elevato, che si abbassa del 34% circa passando da 1 a 2 core e del 9% da 2 a 4 core. Da notare come in tutti i casi del SRJF il tempo di attesa non si sia mai azzerato, o comunque portato vicino allo 0, ma rimasto sopra a valori apprezzabili. Tale dettaglio lo andremo ad esaminare nel prossimo capitolo conclusivo.

Capitolo 5

Conclusioni

Bibliografia

- [1] Repository `github` https://github.com/simo-30/Relazione_finale_laurea_triennale
- [2] Libreria `process.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/processo/process.h
- [3] Libreria `process_list.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/lista_processo/process_list.h
- [4] Libreria `setting.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/setting/setting.h
- [5] Libreria `stat.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/statistics/stat.h
- [6] Libreria `utility.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/utility/utility.h
- [7] Libreria `macro.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/utility/macro.h
- [8] Libreria `fcfs.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/FCFS/fcfs.h
- [9] Libreria `rr.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/RR/rr.h
- [10] Libreria `sjf.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/SJF/sjf.h
- [11] Libreria `srjf.h` https://github.com/simo-30/Relazione_finale_laurea_triennale/blob/master/codice/SRJF/srjf.h