

Sistemi Operativi

Appunti delle Lezioni di Sistemi Operativi

Anno Accademico: 2024/25

Giacomo Sturm

*Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"*

Sorgente del file LaTeX disponibile al seguente link:

<https://github.com/00Darxk/Sistemi-Operativi>

Indice

1	Introduzione e Docker	1
1.1	Comandi di Docker	2
2	Introduzione alla Linea di Comando	3
2.1	Comandi su Directory	3
2.2	Comandi su File	5
2.3	Spostamento e Copia di Elementi	7
2.4	Scrittura di File e Chiamate di Sistema	10
3	Sistemi Operativi	14
3.1	Sistema Operativo come Gestore di Risorse	14
3.2	Processi	18
3.2.1	Creazione di un Processo	19
3.2.2	Divisione in Stati	20
3.2.3	User e Kernel Mode	21
3.2.4	Chiamate di Sistema	22
3.2.5	Ciclo di Vita di un Processo	23
3.2.6	Gestione Efficiente dei Processi	23
3.3	Scheduling	25
3.3.1	First Come, First Served	25
3.3.2	Shortest Job First	25
3.3.3	Shortest Time to Completion First	25
3.3.4	Round Robin	26
3.3.5	Multi-Level Feedback Queue	26
3.3.6	Completely Fair Scheduling	27

1 Introduzione e Docker

Questo corso anche se copre alcuni dei compiti di un amministratore di sistema, non rappresenta un corso di formazione per quest'ultimi. Si tratterà degli aspetti tipici di un kernel di un moderno sistema operativo, trattando nello specifico un sistema Linux, ma saranno applicabili ad ogni sistema operativo moderno. Sarà quindi necessaria una qualsiasi distribuzione Linux. Si utilizza Linux perché è un sistema gratuito e open-source. Si tratta di uno dei sistemi operativi più usati al mondo, usato nella maggior parte dei supercomputer e tra i web server rappresenta il 96% del primo milione in termini di traffico. Installato sul 85% di tutti gli smartphone e conta 1.6 miliardi di utenti. Una distribuzione Linux, o "distro", consiste in un sistema operativo basato sul kernel di Linux, insieme a configurazioni predefinite distribuite da organizzazioni. Esistono centinaia, se non migliaia, di distribuzioni di Linux, ognuna con un suo diverso obiettivo e caratteristiche.

Per utilizzare una distribuzione di Linux si usa il programma Docker Desktop, un software di virtualizzazione leggera. In generale sono possibili due modi di simulare un'applicazione nativa ad un altro sistema operativo. Si può utilizzare una macchina virtuale, che riproduce il sistema operativo da virtualizzare, con tutte le sue librerie, su cui viene eseguita l'applicazione. Tutto questo a sua volta si poggia su un hypervisor per gestire questi sistemi operativi, poggiato a sua volta sul sistema operativo originario della macchina. Questa è una struttura molto profonda e presenta un elevato overhead, causa quindi una perdita di prestazioni.

Negli anni '10 del 2000 si introdusse un meccanismo per alleggerire questo tipo di virtualizzazione, introducendo il concetto di container. Questi contenitori hanno al loro interno solamente l'applicazione nativa al sistema operativo e le sue librerie, senza simulare l'intero sistema operativo. Questi contenitori poggiano direttamente sul sistema operativo base, accompagnati dal software di docker. Questo processo di virtualizzazione leggera è possibile solo in alcuni casi, su Windows e Mac è possibile infatti una gestione semplificata di applicazioni per Linux in questo modo.

Docker rappresenta il software più utilizzato a livello globale per realizzare una virtualizzazione leggera. Tramite docker è possibile interagire direttamente con l'applicazione, anche se è comunque possibile caricare un sistema operativo diverso, in container Linux.

Docker presenta un'architettura a tre livelli, un client permette l'interfaccia con la macchina host, il docker host dove sono contenute le immagini ed i container. Inoltre è presente un registro online di immagini scaricabili. Un'immagine rappresenta un eseguibile con tutte le librerie al completo come container. Un'immagine rappresenta un'applicazione statica, indipendente da file esterni, ed è possibile realizzare un container basato solamente su quell'immagine. Si utilizzano immagini poiché molte applicazioni richiedono versioni diverse delle stesse librerie e file binari, per cui naturalmente solleveranno conflitti, che si vuole evitare.

Un container contiene un'applicazione eseguita in un ambiente isolato, da un'immagine base e le sue librerie e binari, da cui è possibile creare più di un container diversi. Ogni container può avere un volume, uno spazio su disco dove memorizzare dati e configurazioni indipendenti, in uno spazio isolato dall'immagine, per cui è possibile ricaricare l'immagine senza perdere i dati salvati sul volume. Si usa un volume poiché è buona norma che i container non abbiano uno stato, questo viene associato ad uno o più container specificando un mountpoint, accessibile in lettura o scrittura.

Le immagini sono dei template di sola lettura contenenti il root del filesystem che si vuole istanziare, i meta-dati, le variabili d'ambiente, e le varie configurazioni dell'applicazione che si vuole

eseguire. Queste immagini vengono costruite a strati, ognuno di questi può aggiungere, modificare o rimuovere file. Si comincia a creare un'immagine partendo da un contenitore vuoto ed aggiungendo progressivamente nuovi strati l'uno sull'altro. Tutti questi strati sono in sola lettura, solamente l'ultimo strato di default è anche in scrittura, per aggiungere configurazioni o file finali, anche se si preferisce inserirli in un volume con un ciclo di vita isolato.

Utilizzando container, immagini e volume è possibile disaccoppiare tutte le componenti del sistema operativo, le applicazioni ed i dati, rispettivamente.

1.1 Comandi di Docker

La sintassi di Docker venne ripulita ed aggiornata nel 2017, per cui risulta molto semplice ed intuitiva. Il comando principale è il prefisso **docker**, si possono scegliere operazioni sulla gestione di container, immagini o volumi specificando la parola chiave **container**, **image** o **volume**. Dopo di questi si possono inserire tipici comandi come **ls** per visualizzare gli elementi contenuti, rimuoverli con **rm**, etc. Si utilizza anche la parola chiave **network** per gestire la rete di container.

Il comando **help** fornisce una lista di comandi disponibili.

In questo corso si utilizzerà l'immagine, presente nel docker hub, **chmod84/os2425**.

Per creare un container sulla base di questa immagine, bisogna scaricarla con i seguenti comandi:

```
prompt> docker image pull chmod84/os2425
```

Questo comando di default prende l'ultima versione dell'immagine disponibile, in caso si voglia utilizzare una versione specifica si indica con : seguito dal numero di versione oppure **latest**, per l'ultima versione disponibile. Questo tag viene associato dal produttore della versione arbitrariamente, quindi potrebbe non corrispondere all'ultima versione esistente dell'immagine. In seguito per creare ed avviare il container si usa il seguente comando:

```
prompt> docker container --name os2425  
--hostname sistemi-operativi -v dati:/corso -it chmod84/os2425
```

Questo comando crea un volume chiamato "corso" nella root del filesystem.

Per avviare o terminare un container si usano i comandi:

```
prompt> docker container start os2425  
prompt> docker container stop os2425
```

Per aggiornare l'immagine bisogna prima rimuovere il vecchio container:

```
prompt> docker container rm os2425
```

In seguito si effettua una pull della nuova immagine e si crea il container come descritto precedentemente. Per avviare la shell di un container precedentemente avviato si utilizza il seguente comando:

```
prompt> docker container exec -it os2425 /bin/bash
```

2 Introduzione alla Linea di Comando

All'apertura del container docker ci si trova davanti ad una riga di comando del tipo:

```
corso@sistemi-operativi:~$
```

Il termina prima della chiocciola indica l'utente con cui si è effettuato l'accesso. La maggior parte dei sistemi operativi moderni infatti sono sviluppati su di una base multiutente. Per cui lo stesso sistema può essere utilizzato da utenti diversi, anche contemporaneamente con una separazione di privilegi, isolando i file appartenenti ai vari profili degli utenti. Si utilizza quindi l'utente **corso**. Mentre il nome dell'host è **sistemi-operativi**, specificato dopo la chiocciola. Un'ulteriore separatore **:** lo separa dai caratteri **~\$**. Il primo rappresenta un percorso, descritto successivamente, indica la directory corrente, in questo caso **~** indica la cartella di base dell'utente utilizzato per sfruttare il sistema. Il carattere **\$** indica dove viene inserito il comando o l'applicativo sulla riga di comando. Al posto di questo carattere può essere presente un cancelletto **#**, nel caso di livello di privilegio di amministratore.

Quando viene effettuato un comando, viene restituito un output a schermo, ed alla riga successiva, viene riproposto il prompt dei comandi. Non necessariamente tutti i comandi producono un output visivo a schermo.

2.1 Comandi su Directory

Il comando **pwd** "Print Working Directory" stampa a schermo la directory corrente dove si sta lavorando.

```
corso@sistemi-operativi:~$ pwd
/home/corso
corso@sistemi-operativi:~$
```

Questo concetto è fondamentale, poiché mostra in quale posizione del filesystem vengono eseguiti i comandi proposti, all'interno del quale vengono effettuate le operazioni, che possono essere di creazione, modifica, rimozione, di file o di altre directory. L'indirizzo fornito si legge da sinistra verso destra, e specifica la root del filesystem **/**, dove si trova la cartella **home**, dove si trova la cartella **corso**, dove ci troviamo attualmente. Il filesystem è un oggetto di estrema complessità, che permette di astrarre in maniera semplice la virtualizzazione dei dati. Questo permette all'utente di interfacciarsi direttamente con l'hardware del calcolatore. Questo sistema organizza i file in cartelle, contenenti altre cartelle, identificate da un percorso o directory, per individuarli in modo univoco. Rappresenta il percorso dalla radice dell'albero al nodo specifico individuato nel filesystem. Le cartelle possono essere espresse equivalentemente sia come **cartella/** che **cartella**, si utilizza **/** dopo il nome della directory per specificare si tratti di una cartella e non di file. Se un nodo ha dei figli, allora si tratta di una cartella, mentre tutti i file sono foglie, ma non tutte le foglie sono file, e non possono essere espressi con uno **/** finale.

Per spostarsi all'interno del filesystem, è disponibile il comando **cd** "Change Directory", seguito dal percorso:

```
corso@sistemi-operativi:~$ cd /home
corso@sistemi-operativi:/home$
```

Per ritornare alla directory iniziale è sufficiente effettuare nuovamente un `cd`:

```
corso@sistemi-operativi:/home$ cd corso
corso@sistemi-operativi:~$
```

Se viene eseguito correttamente, non produce alcune output, mentre se riscontra un errore viene mostrato a schermo.

Esistono essenzialmente due tipi di percorsi, relativi ed assoluti. Si distinguono semplicemente per la presenza di uno `/` iniziale. I percorsi assoluti specificano l'intero percorso nella sua interezza, partendo dalla radice del filesystem, come `/home/corso`. I percorsi che non iniziano con `/` sono percorsi relativi, come `corso`, all'interno di `/home`.

Se si utilizza un percorso assoluto, l'interprete dei comandi ci porta esattamente all'interno di quel percorso, altrimenti l'interprete dei comandi concatena la working directory con il comando relativo inserito, generando il percorso da seguire. Non c'è alcuna differenza nel funzionamento di questi due comandi, poiché viene gestito trasparentemente dal sistema operativo.

Ogni directory contiene sempre due directory speciali: `.` e `..`, dove `.` indica la directory corrente. Per cui `/home/corso/.` è uguale a `/home/corso`. Sarà molto utile in comandi successivi. La directory `..` rappresenta la directory esattamente superiore nel filesystem. Per cui `/home/corso/..` è uguale a `/home`. Quest'ultima directory ci permette di risalire l'albero del filesystem gradualmente. Viene utilizzata per semplificare il movimento all'interno del filesystem, senza necessariamente conoscere tutti i percorsi possibili alle directory superiori. Anche la radice contiene queste due cartelle, quindi sembrerebbe possibile salire al livello superiore della radice, ma il comando `cd ..` riporta alla radice.

La home directory è dove vengono archiviati tutti i file di un determinato utente, in generale in un sistema Linux, ogni utente ha una sua directory all'interno della directory `/home`, chiamata con il proprio nome utente. Utilizzare nomi differenti è possibile, ma non avviene di default. In ogni posizione del filesystem è possibile utilizzare il comando `cd ~` per spostarsi direttamente nella home directory, ma questo simbolo non è presente nelle tastiere italiane, bisognerebbe utilizzare infatti la combinazione ASCII "Alt+126". Tuttavia è possibile effettuare la stessa operazione con `cd` senza argomenti.

Per creare delle directory esiste il comando `mkdir` "Make Directory", seguito dal percorso che si vuole realizzare, indicato con un percorso relativo o assoluto. Tutte le cartelle precedenti all'ultima devono esistere, per permettere di creare la cartella.

```
corso@sistemi-operativi:~$ mkdir tutorial
corso@sistemi-operativi:~$
```

Equivalente a:

```
corso@sistemi-operativi:~$ mkdir /home/corso/tutorial
corso@sistemi-operativi:~$
```

Si possono creare più cartelle contemporaneamente aggiungendo un altro percorso separato da spazi:

```
corso@sistemi-operativi:~$ mkdir dir1 dir2 dir3
corso@sistemi-operativi:~$
```

Esiste un'opzione che si può passare a `mkdir` per creare un intero percorso in una volta sola tramite `-p`:

```
corso@sistemi-operativi:~$ mkdir -p dir4/dir5/dir6
corso@sistemi-operativi:~$
```

Se questo comando viene chiamato senza argomento, stampa un errore a schermo, poiché richiede un argomento, una cartella da creare. Non si vuole memorizzare il contenuto di tutte le cartelle del filesystem, si utilizza quindi il comando `ls` "List". Questo comando stampa a schermo la lista di tutti i file o cartelle contenute nella working directory:

```
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir3 dir4
corso@sistemi-operativi:~$
```

Si può passare come argomento un percorso nel filesystem per vedere i suoi contenuti:

```
corso@sistemi-operativi:~$ ls dir4
dir5
corso@sistemi-operativi:~$
```

Poiché due percorsi vengono separati da spazi, non sembra possibile poter creare una cartella che abbia un nome contenente spazi, un'opzione è utilizzare un underscore `_` al posto degli spazi nella stringa. Poiché questo gestisce la stringa come una singola parola, se si vogliono mantenere le parole differenti bisogna quindi inserire lo spazio nella stringa passata come argomento. Per poter inserire il carattere corrispondente allo spazio, si può incapsulare la stringa che si vuole utilizzare con singoli o doppi apici: " ... " o ' ... '. Oppure si può utilizzare il carattere `\` seguito dallo spazio: ... \ Questo si chiama carattere di escape, ed indica che il carattere che viene dopo non deve essere interpretato dall'interprete dei comandi, ma appartiene alla stessa stringa che si sta inserendo. Questo carattere di escape è comune per molti linguaggi di programmazione quando si vogliono inserire caratteri speciali nelle stringhe. In generale se sono contenenti spazi in una stringa, questa viene mostrata tra singoli apici.

2.2 Comandi su File

Oltre a gestire directory, il filesystem gestisce file regolari, ed è possibile crearli e modificarli analogamente alle directory. Il sistema operativo per mostrare a video un'informazione utilizza lo "Standard Output" o `stdout`, collegata al terminale, in modo da mostrare il risultato di ogni comando. La shell quindi legge il contenuto del file dello `stdout` e lo mostra a video. Poiché i comandi stampano il loro output su questo file virtuale, è possibile stampare gli output su di un file reale, presente nel filesystem. Utilizzando l'operatore di redirectione `>` invia l'output di un comando ad un file, indicando il suo percorso:

```
corso@sistemi-operativi:~$ ls > output.txt
corso@sistemi-operativi:~$
```

Se il file non esiste, viene creato all'invocazione del comando:

```
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir3 dir4 output.txt
corso@sistemi-operativi:~$
```

Si può esplicitare l'estensione del file, ma anche senza l'interprete dei comandi è in grado di creare un file di caratteri ASCII, quindi un file testuale. Per leggere il contenuto di un file quindi è possibile inviare il suo contenuto allo stdout. Uno dei comandi per effettuare questo è `cat`, "Concatenata". Prende come argomento la lista dei file di cui si vuole stampare il contenuto a schermo, e produce una concatenazione dei contenuti di questi file:

```
corso@sistemi-operativi:~$ cat output.txt
tutorial
dir1
dir2
dir3
dir4
output.txt
corso@sistemi-operativi:~$
```

Se si utilizza il comando `ls` su di un file, questo cerca se esiste il file che stiamo cercando, mostrando il suo nome a schermo:

```
corso@sistemi-operativi:~$ ls output.txt
output.txt
corso@sistemi-operativi:~$
```

In realtà stampa il percorso relativo che è stato utilizzato come argomento, poiché il file che stiamo cercando si trova nella working directory, il percorso assoluto di questo output è `/home/corso/output.txt`.

L'operatore di redirectione rappresenta un modo abbastanza comune per creare file contenenti il risultato di comandi, o programmi articolati. Per creare dei file arbitrari, si può utilizzare il comando `echo`, insieme ad un operatore di redirectione. Questo comando infatti stampa a video quello che gli viene passato come argomento tra doppi apici:

```
corso@sistemi-operativi:~$ echo "test"
test
corso@sistemi-operativi:~$
```

Tramite l'operatore di redirectione si possono quindi realizzare file arbitrari:

```
corso@sistemi-operativi:~$ echo "test" > test
corso@sistemi-operativi:~$
```


L'uso esplicito di un'estensione, risale agli anni '80 dove i file venivano differenziati rispetto alla loro estensione, ma i sistemi operativi moderni sono in grado di determinare il contenuto di un file e quindi la sua estensione. Il comando chiamato `file`, seguito dal file di cui si vuole conoscere il tipo:

```
corso@sistemi-operativi:~$ file output.txt
output.txt: ASCII text
corso@sistemi-operativi:~$ file test
test: ASCII text
corso@sistemi-operativi:~$
```

L'operatore di redirectione, quando viene utilizzato su di un file, ne tronca il contenuto, lo azzerava, per poi scrivere quello che viene passato allo `stdout`, dal primo byte del file. L'operatore di redirectione `>>` effettua la stessa operazione di `>`, ma scrive all'interno del file senza troncarlo, nel suo spazio vuoto, appendendo il contenuto nuovo alla fine:

```
corso@sistemi-operativi:~$ echo "test" >> output.txt
corso@sistemi-operativi:~$ cat output.txt
tutorial
dir1
dir2
dir3
dir4
output.txt
test
corso@sistemi-operativi:~$
```

Se si ha un file contenente un numero elevato di righe o parole, questo non permette di visualizzarlo interamente in una singola schermata. Si può scrollare per visualizzare l'intero file, poiché è contenuto in una finestra grafica che bufferizza tutto quello che succede. Ma per alcune funzioni questo non avviene, quindi si vuole poter determinare le proprietà di un file, senza doverlo visualizzare nella sua integrità. Il comando `less` prende come argomento un percorso, ed alla sua invocazione, trasforma l'interfaccia grafica in un programma di visualizzazione del file, in cui si può scorrere tra le sue righe utilizzando le frecce verso l'alto ed il basso, quando non è disponibile un'interfaccia grafica che permette di scrollare sul file. Questo comando è erede di un altro chiamato `more`, che realizza la stessa operazione in modo più grezzo, con meno funzionalità. Per ritornare alla riga di comando bisogna digitare il tasto `q`. Questo programma offre un suo prompt, all'ultima riga dell'interfaccia grafica indicato da `:`, dove possono essere eseguite operazioni sul file aperto.

2.3 Spostamento e Copia di Elementi

Per spostare dei file si utilizza il comando "Move" `mv` seguito da una serie di argomenti, dove l'ultimo rappresenta il percorso dove vengono spostati i file o directory indicate nei precedenti argomenti:

```
corso@sistemi-operativi:~$ mv output.txt dir1
corso@sistemi-operativi:~$ ls dir1
```

```
output.txt
corso@sistemi-operativi:~$
```

Per riportarlo nella directory corrente, si può specificare per intero il percorso di questa, oppure si può utilizzare la directory .:

```
corso@sistemi-operativi:~$ mv dir1/output.txt .
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir3 dir4 output.txt test
corso@sistemi-operativi:~$
```

Questo rappresenta uno degli utilizzi di queste due cartelle . e .. per scrivere comandi in modo estremamente sintetico. Un modo analogo per effettuare quest'operazione consiste nell'usare ...:

```
corso@sistemi-operativi:~$ cd dir1
corso@sistemi-operativi:~/dir1$ mv output.txt ..
corso@sistemi-operativi:~/dir1$ cd
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir3 dir4 output.txt test
corso@sistemi-operativi:~$
```

Quando si utilizza una sequenza di argomenti per il comando mv, tutti gli argomenti eccetto l'ultimo sono gli elementi da spostare, mentre l'ultimo indica la destinazione di questi elementi:

```
corso@sistemi-operativi:~$ mv output.txt test dir3 dir2
corso@sistemi-operativi:~$ ls dir2
output.txt test dir3
corso@sistemi-operativi:~$
```

Se viene inserito come ultimo elemento un file, produce un errore.

Il comando cp "Copy", copia un file di origine, in un percorso specificato del filesystem:

```
corso@sistemi-operativi:~$ cp dir2/output.txt .
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir4 output.txt
corso@sistemi-operativi:~$
```

Quando si effettua sulla stessa directory contenente il file origine, bisogna specificare un nuovo nome per questo file, altrimenti si rischierebbe di sovrascriverlo. Si realizza specificando un altro file, che il sistema crea all'invocazione del comando:

```
corso@sistemi-operativi:~$ cp output.txt output\ copy.txt
corso@sistemi-operativi:~$ ls
tutorial dir1 dir2 dir4 output.txt 'output copy.txt'
corso@sistemi-operativi:~$
```

In una singola operazione quindi è possibile creare una copia di un file in una directory, scegliendo un nome differente. Utilizzando il comando mv è possibile rinominare un file, specificando come ultimo argomento il suo nuovo nome:

```
corso@sistemi-operativi:~$ mv dir2/test dir2/test2
corso@sistemi-operativi:~$ ls dir2
output.txt test2 dir3
corso@sistemi-operativi:~$
```

Quindi se l'ultimo argomento è un file, il comando non può supportare una lista di elementi da spostare. È possibile rinominare allo stesso modo una cartella, mantenendo gli elementi contenuti:

```
corso@sistemi-operativi:~$ mv dir2 dirX
corso@sistemi-operativi:~$ ls dirX
output.txt test2 dir3
corso@sistemi-operativi:~$ mv dirX dir2
corso@sistemi-operativi:~$ ls dir2
output.txt test2 dir3
corso@sistemi-operativi:~$
```

Per rimuovere file, o cartelle in certe condizioni, si può utilizzare il comando "Remove" `rm`, seguito dal percorso dove si trova l'elemento da eliminare:

```
corso@sistemi-operativi:~$ rm dir2/test2
corso@sistemi-operativi:~$ ls dir2
output.txt dir3
corso@sistemi-operativi:~$
```

Questo comando non richiede conferme, quindi non è possibile correggere l'errore, come è possibile in sistemi operativi analoghi a Windows, dove è disponibile una cartella "Cestino" dove sono contenuti gli elementi eliminati. Per rimuovere le directory invece si può utilizzare il comando `rmdir`, "Remove Directory":

```
corso@sistemi-operativi:~$ rmdir dir1
corso@sistemi-operativi:~$ ls
tutorial dir2 dir4 output.txt 'output copy.txt'
corso@sistemi-operativi:~$
```

Tuttavia questo non può essere effettuato per directory non vuote:

```
corso@sistemi-operativi:~$ rmdir dir2
rmdir: failed to remove 'dir2': Directory not empty
corso@sistemi-operativi:~$
```

Allora per rimuovere questa cartella, si potrebbero rimuovere manualmente tutti i suoi file, risalendo fino a `dir2`. Esiste un'opzione che semplifica questo procedimento, effettuando una rimozione ricorsiva, rimuovendo tutti gli elementi contenuti all'interno del percorso specificato, prima di rimuovere gli elementi superiori. Quest'opzione di `rm` scende fino alle foglie, dato il percorso specificato, e le rimuove, risalendo nell'albero fino a quando non è in grado di rimuovere tutto il percorso passato come argomento.

2.4 Scrittura di File e Chiamate di Sistema

Si può utilizzare un editor di testo nativo alla command line, accessibile tramite il comando **nano**, per poter creare file C.

```
corso@sistemi-operativi:~$ nano hello.c
```

All'interno dell'editor, in basso sono presenti gli shortcut disponibili, per salvare un file si utilizza il comando "Ctrl+O", ed in seguito si inserisce il nome del file e si conferma con invio. Per uscire dall'editor si utilizza il comando "Ctrl+X".

Si crea questo file dove è presente una singola istruzione di stampa della stringa **Hello World**:

```
#include <stdio.h>

int main(){
    printf("Hello World\n");
}
```

Sono compilabili con la stessa sintassi dei precedenti codici, tramite l'interprete gcc:

```
corso@sistemi-operativi:~$ gcc -o hello hello.c
corso@sistemi-operativi:~$ ./hello
Hello World
corso@sistemi-operativi:~$
```

Invece di scrivere direttamente il codice sul container, è possibile trasferire un file scritto sulla macchina host, con il comando **cp** seguito dal nome del file da copiare sul container. Bisogna anche specificare dove bisogna salvare il file, si indica il nome del container e dopo i due punti il percorso assoluto al suo interno:

```
prompt> docker cp hello.c os2425:/home/corso
```

Dentro la directory **/ostep-code/** sono presenti molti codici con esempi di 3.2.4, analizzate in questo corso.

Il comando **man** fornisce le pagine di manuali relativi alla chiamata di sistema o comando forniti come argomenti. Ognuna di queste pagine di manuale contiene una sezione che specifica i possibili valori di ritorno di questa chiamata di sistema. Questo manuale è disponibile in ogni sistema Linux, e rappresenta una delle funzioni più utili disponibili in un sistema operativo, poiché fornisce tutte le informazioni necessarie per utilizzare le chiamate di sistema nella programmazione.

La maggior parte delle pagine relative a chiamate di sistema sono divise in sezioni. Una prima sezione fornisce una descrizione sintetica del suo funzionamento, e viene descritta la sua sintassi, e le librerie necessarie per essere utilizzate. Molte di queste chiamate di sistema restituiscono un valore di tipo **pid_t**, essenzialmente un intero, ma fornisce informazioni aggiuntive quando viene riferito a dei processi. Segue una descrizione estesa della chiamata. Ognuna di queste pagine contiene una sezione per i valori di ritorno, molto utili per interpretare il comportamento della funzione. Per scorrere tra le varie pagine del manuale di una funzione si utilizza il tasto **F**, mentre per chiudere il manuale si utilizza il tasto **Q**.

Poiché è un elemento molto complesso, esiste una pagina di manuale dedicata ad esso stesso, che spiega come è diviso ed in quale capitoli trovare quello che si sta cercando. Questa suddivisione permette di risolvere omonimie. Per accedere alle pagine di manuale dedicate ad una di queste funzioni omonime, se non si vuole accedere alla prima, si inserisce prima del nome della chiamata di sistema.

Alcune di queste chiamate di sistema, non sono vere e proprie chiamate di sistema, ma rappresentano delle versioni “wrapped” di vere chiamate di sistema.

Per spostare la cartella di interesse si utilizza la flag `-r` sul comando `cp` che copia ricorsivamente una cartella:

```
corso@sistemi-operativi:~$ cd /ostep-code/
corso@sistemi-operativi:/ostep-code$ cp -r cpu-api/ /home/corso
corso@sistemi-operativi:~$
```

Nel file `p1.c` sono presenti delle semplici chiamate di sistema:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

La funzione `getpid()` restituisce il “Process ID”, `pid`, del processo che l’ha invocato. La funzione `fork()` crea un’altra istanza dello stesso processo, duplicando la memoria, in uno spazio separato, ma ha lo stesso contenuto, tranne per il valore di ritorno di questa funzione. Per il processo padre è il `pid` del padre, mentre per il figlio è esattamente 0. Per entrambi i processi la prima istruzione da eseguire è la prima istruzione seguente al `fork`.

Questo programma stampa il `pid` di questi due processi, verificando con un’istruzione condizionale il loro valore di ritorno. Non rappresenta un’esecuzione tramite threads, poiché non avviene sullo stesso spazio di memoria, e non sono presenti variabili condivise, che potrebbero generare conflitti, analizzati successivamente. Poiché questi processi vengono eseguiti su spazi di memoria isolati tra di loro, non per scambiarsi dati tra di loro devono utilizzare ulteriori chiamate di sistema come per generare collegamenti come “pipe” o “socket”.

Quando viene eseguito questo file `p1.c` non è garantito che venga sempre eseguito prima il processo padre e poi il processo figlio. Non è possibile conoscere a priori in termini generali come lo scheduler decide di gestire il processo padre ed il processo figlio. Nella vita reale potrebbe essere richiesto sincronizzare due processi, quindi bisognerebbe poter controllare il quando vengono eseguiti rispetto agli altri. Per risolvere questo problema ed inserire un ordine certo e noto a priori a questo output, si utilizza la funzione `wait()` per introdurre un meccanismo di attesa.

Nel file `p2.c` è contenuta la chiamata di sistema `wait()`, la funzione attende il cambiamento di stato di un processo, non solo la sua terminazione:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1)
    } else { // parent goes down this path (original process)
        int wc = wait(NULL)
        printf("hello, I am parent of %d (wc: %d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

Questa funzione prende come argomento un puntatore ad intero di tipo `Nullable`, quindi si può utilizzare il puntatore `null`, se non si vuole catturare il valore di ritorno. Esistono tre varianti della funzione `wait()`, `waitpid()` e `waitid()`. Il valore restituito dalla funzione è di tipo `pid_t`, o `int`, solo per `waitid()`, in caso di terminazione rappresenta l'id del processo terminato, oppure -1 se la funzione ha riscontrato errori o ha fallito. La prima funzione attende il cambiamento di un qualsiasi processo, la seconda attende il cambiamento di un processo specifico.

Queste funzioni vengono utilizzate per osservare il cambiamento di stato dei figli nel processo chiamante. Un cambiamento di stato sono la terminazione, l'aver ricevuto un segnale ed avere interrotto l'esecuzione o continuato l'esecuzione. Quando un figlio viene terminato, tramite una `wait` è possibile liberare le risorse associate al processo figlio. La differenza rispetto all'esecuzione di `p1.c` è che il processo padre non può stampare la sua riga di testo se non è stato eseguito il processo figlio, tramite una funzione `wait()`.

Per scrivere all'interno di un file, da creare o già esistente, si utilizzano tre chiamate di sistema, la `open()` per aprire il file e permettere di modificarlo, la `write()` per scrivere all'interno del

file, e si utilizza la `close()` per chiudere il file aperto. Dalle pagine di manuale si osserva che la funzione `open()` necessita della libreria `fcntl.h` e richiede come argomento una stringa che contiene il percorso al file che si vuole aprire, o creare. Si può specificare inoltre il suo comportamento, e bisogna specificare la modalità del file, ovvero i suoi permessi, in caso si decida di crearlo. Si può concatenare la scrittura, troncare il file, oppure crearne uno nuovo, in base alle flag specificate nella `open()`. Queste flag possono essere concatenate insieme, tramite l'operatore `|`. Si utilizzano le seguenti flag per indicare di troncare il file in caso esista, altrimenti di crearlo e di assegnare privilegi di lettura e scrittura al possessore dello stesso. Inoltre si specifica la modalità di accesso: in solo lettura, solo scrittura, oppure lettura e scrittura. In questo caso si apre in modalità di sola scrittura:

```
int fd = open("test.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRWXU);
```

Quando un file viene aperto in un sistema Linux, il sistema operativo mette a disposizione un riferimento al file, un "File Descriptor", contenuto in un intero. Nelle seguenti operazioni si userà il FD, invece del percorso del file, per accedere al file aperto. Se non viene specificato il percorso del file, viene creato nella working directory.

Per effettivamente scrivere dentro al file si utilizza la funzione `write()`, che prende come argomento il FD, un buffer da cui prende cosa scrivere, ed un intero che specifica fino a quando deve continuare a scrivere byte per byte. Questa funzione richiede dell'ulteriore libreria `unistd.h`:

```
char* txt = "test";
write(fd, txt, strlen(txt));
```

Dopo aver effettuato queste modifiche si può chiudere il file tramite la `close()`, dato il FD. Si ha quindi un file completo del tipo:

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int main(){
    int fd = open("test.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRWXU);

    char* txt = "test";
    write(fd, txt, strlen(txt));

    close(fd);
}
```

3 Sistemi Operativi

Quando un programma viene eseguito segue sempre una serie di passaggi, comuni per ogni programma in esecuzione su un processore, chiamato ciclo di fetch-decode-execute, analizzato dettagliatamente nel corso di calcolatori elettronici:

- Fetch: il processore preleva l'istruzione da eseguire dal programma;
- Decode: l'istruzione viene trasformata in una o più istruzioni macchina (in base al tipo di architettura del processore);
- Execute: l'istruzione macchina viene effettuata dal processore;
- Il processore aumenta il PC e passa alla prossima istruzione.

Questo sistema si basa sul modello di Von Neumann, ma le prestazioni e le aspettative per calcolatori moderni sono molto maggiori di questo, una sostanziale differenza per le macchine moderne è la capacità di eseguire più di un programma in simultanea.

Il sistema operativo è ciò che permette ai calcolatori moderni di poter effettuare queste esecuzioni simultanee, quindi si trova responsabile della gestione di tutti i programmi in esecuzione su una macchina. Ha il compito di rendere semplice l'esecuzione di un programma, permettendogli di condividere memoria con tutti gli altri processi in esecuzione, e permettergli di interagire con periferiche o dispositivi di I/O. Il compito del sistema operativo consiste nell'effettuare queste operazioni efficientemente e velocemente.

3.1 Sistema Operativo come Gestore di Risorse

Il sistema operativo effettua un'operazione di virtualizzazione, trasformando le componenti fisiche del calcolatore in una forma virtuale, indipendentemente, in modo che questi programmi siano in grado di interagirci. Ha la funzione di connettere questi programmi e l'hardware dove vengono eseguiti. Le risorse fisiche sono il processore, la memoria condivisa, il disco, le periferiche, ed altri dispositivi connessi. Questa forma virtuale è quindi più accessibile ai programmi e di più facile uso e generale. A volte quindi ci si può riferire al sistema operativo come una macchina virtuale, capace di creare versioni virtuali dei componenti del calcolatore.

Queste funzionalità introdotte dal sistema operativo possono venire invocate da parte di un qualsiasi programma, ma non è detto che a questo venga permesso di effettuare tutte le operazioni offerte. Tratteremo questo concetto nella sezione sui processi. Ai programmi sono disponibili delle chiamate di sistema, invocazioni di funzioni specifiche offerte dal sistema operativo sulle componenti virtuali, in modo che questo possa interfacciarsi direttamente con l'hardware, agendo come un intermediario per i programmi. In generale il sistema operativo offre delle librerie standard o API per potersi interfacciare con le chiamate di sistema. Queste chiamate di sistema consistono nell'eseguire programmi, accedere alla memoria e dispositivi di I/O, tra le più comuni ed utilizzate, ma in generale fornisce sulle centinaia di chiamate di sistema.

Il sistema operativo si comporta come un gestore di risorse, come il processore, la memoria ed il disco. Si definisce gestore poiché sceglie come condividere il processore a tutti i processi in esecuzione, contemporaneamente in accesso alla stessa memoria, ed allo stesso disco e periferiche.

Un calcolatore moderno dispone di un certo numero di processori fisici, o virtuali. Contengono una decina di core fisici, ma una macchina server potrebbe disporre di decine di core sparsi su più processori. Generalmente il numero di programmi in esecuzione è molto maggiore del numero di core o processori disponibili dalla macchina. Il sistema operativo è quindi in grado di fornire l'illusione di avere un numero molto elevato di core, per ciascuno dei processi in esecuzione, grazie alla virtualizzazione delle risorse.

Si considera un semplice esempio fornito dal docente nel file `cpu.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[]){
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1); // Repeatedly checks the time and
        returns once it has run for a second
        printf("%s\n", str);
    }
    return 0;
}
```

Questo programma prende come argomento una stringa di caratteri e ripete costantemente dopo ogni secondo il primo carattere della stringa. Si può terminare con la combinazione di tasti “Control + C”. Dopo aver compilato in un file binario il codice, per eseguirlo si indica il suo indirizzo relativo, ed il sistema operativo automaticamente lo interpreta come un programma da eseguire, quindi non rappresenta un comando per eseguire direttamente il programma:

```
corso@sistemi-operativi:~$ gcc -o cpu cpu.c
corso@sistemi-operativi:~$ ./cpu "A"
```

Inoltre si possono eseguire più di un programma contemporaneamente combinando i comandi con un `&`:

```
corso@sistemi-operativi:~$ ./cpu "A" & ./cpu "B" & ./cpu "C" & ./cpu "D"
```

Tutti questi processi sembrano essere eseguiti allo stesso tempo, si definisce processo un programma in esecuzione, ma l'ordine in cui vengono effettuate le istruzioni di ogni programma non è costante.

La memoria fisica è composta da un array di byte, ed un programma mantiene tutte le sue informazioni su strutture di dati in memoria. Divise in memoria in scrittura (store), specifica i dati da scrivere al dato indirizzo, e lettura (load), specificando l'indirizzo per accedere ai dati.

Si utilizza un altro programma offerto dal docente per poter analizzare il comportamento della memoria virtuale, chiamato `mem.c`:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    int *p = malloc(sizeof(int)); // a1: allocate some memory
    assert(p != NULL);
    printf("(d) address of p: %08x\n",
        getpid(), (unsigned) p); // a2: print out the address of the memory
    *p = 0; // a3: put zero into the first slot of the memory
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

Ogni istanza del programma alloca memoria allo stesso indirizzo, e sembra aggiornare i valori allo stesso indirizzo, indipendentemente.

Eseguendo più istanze di questo programma si rivela come ogni istanza del programma abbia una sua memoria virtuale dedicata, in realtà è il sistema operativo che mappa tra questi spazi di indirizzi virtuali e la memoria fisica. In questo modo si evitano interferenze tra processi, proteggendo la stabilità e sicurezza tra i vari processi.

Questi programmi vengono eseguiti essenzialmente in contemporanea, per cui il sistema operativo deve gestire il processore per fornire l'illusione che tutti questi programmi vengono eseguiti allo stesso tempo. Questo si chiama problema della concorrenza. Programmi moderni esacerbano questo necessitando di più thread, necessitando di più core virtuali.

Si considera il seguente codice `thread.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int counter = 0;
int loops;
```

```
void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
        counter++;
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads<value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```

Questo programma utilizza due threads per modificare una variabile condivisa, incrementandone il valore da zero ad ogni ciclo. Per un numero di cicli relativamente piccolo, il valore restituito da questo programma rappresenta esattamente il doppio del valore inserito come argomento. Ma per numeri più elevati di cicli di loop, il valore tende a cambiare rispetto al valore atteso. Il problema dipende dal modo in cui viene incrementato il contatore condiviso. L'incremento del contatore non corrisponde ad un'istruzione atomica, ma è composta da tre operazioni distinte, la lettura ed il carico del valore attuale su un registro, l'incremento, ed il trasferimento in memoria. Queste tre operazioni non vengono eseguite atomicamente, da cui sorge il problema della concorrenza. Se due thread eseguono quest'operazione simultaneamente allora, è possibile che uno di questi legga il valore che l'altro ancora non ha avuto tempo di trasferire in memoria prima dell'incremento. Questo fenomeno è noto come condizione di gara, dove il risultato dipende dall'ordine in cui i thread accedono alle risorse condivise. Per risolvere questi problemi possono introdurre meccanismi di lock o di blocco, per sincronizzare i due thread, in modo che solo unthread alla volta sia in grado di accedere alla risorsa condivisa.

Quando si utilizzano questi meccanismi evitare di utilizzarli in modo improprio, altrimenti è possibile che sorgano altri problemi di deadlock, dove i thread aspettano all'infinito lo sblocco di una risorsa condivisa.

Dispositivi di memoria condivisa come DRAM memorizzano i dati in modo volatile, per l'esecuzione dei programmi. Ovvero perdono i valori memorizzati quando il sistema viene spento. Quindi si ha la necessità di memorizzare i dati in maniera persistente grazie a sistemi hardware e software. I sistemi utilizzano sistemi I/O come dischi a stato solido (SSD) o dischi rigidi (HDD),

non volatile. Dal punto di vista del software invece, per la persistenza il filesystem gestisce il disco ed è responsabile per memorizzare tutti i file che verranno creati dall'utente. Questo gestore alloca lo spazio necessario sul disco per il file, organizza i suoi dati in modo da essere disponibile in futuro, e corregge e gestisce eventuali errori che potrebbero verificarsi nel processo di scrittura.

Si considera il seguente codice per la creazione di un file con scritto "Hello World" alla directory /tmp/file del container Linux:

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

Le funzioni `open()`, `write()` e `close()` vengono indirizzate alla parte del sistema operativo chiamato filesystem che gestisce questo tipo di richieste.

Questa parte del sistema operativo si occupa di strati aggiuntivi di astrazione come dischi RAID, gestori di volumi logici, etc. Trova una posizione sul disco dove i nuovi dati risiederanno. Infine invia richieste di I/O ai dispositivi di memoria specifici. Il filesystem si occupa degli errori in scrittura che possono sorgere durante la scrittura su disco o guasti ai dispositivi. Utilizza due diversi meccanismi il "journaling" o "copy-on-write", e gestendo ed ordinando file in maniera efficiente sul disco. La tecnica di journaling consiste nel mantenere un registro di tutte le operazioni di scrittura, così da essere in modo ci completarle o ripristinare lo stato in caso di guasti al sistema. La tecnica di copy-on-write invece consiste nella creazione di una copia, invece di sovrascriverli alla modifica dei dati. In questo modo si evitano corruzioni dei dati durante la scrittura, nella nuova posizione.

3.2 Processi

Un processo viene definito come un qualsiasi programma in esecuzione, dove un programma è un agglomerato di istruzioni isolati su una zona di memoria secondaria. Il sistema operativo ha il compito di caricare questo programma ed eseguirlo.

Per dare l'illusione di un'esecuzione simultanea di molti processi, il sistema operativo assegna ad ogni processo un quanto di tempo per cui quel processo può essere eseguito dal processore. Questi processi in esecuzione vengono poi interrotti tramite dei segnali di interrupt, ed il sistema operativo assegna il prossimo processo in coda ad essere eseguito. Il sistema operativo quindi genera una coda di processi da essere eseguiti in base in base a politiche diverse, trattate successivamente

nella sezione dedicata allo scheduling. Questo processo di condivisione del tempo si chiama “time sharing”.

L’operazione di terminare preventivamente l’esecuzione di un processo per poi passare ad eseguire un altro si chiama “context switching”. Questo fornisce l’illusione di avere molti processori virtuali dedicati ad ogni processo. Ma avere molti processi in esecuzione comporta quanti di tempo di lunghezza minore, assegnati a ciascun processo. Con un numero infinito di processi infatti, i quanti di tempo sarebbero di tempo infinitesimo, ed il sistema operativo sarebbe costantemente occupato ad effettuare l’operazione di context switching.

Quest’operazione viene effettuata dal sistema operativo indipendentemente dall’utente, la politica di gestione è salvata in un file sorgente specifico, non accessibile all’utente.

In generale un processo può essere caratterizzato dal suo stato di esecuzione, tutto ciò che questo può leggere o scrivere in esecuzione. È caratterizzato dal suo spazio di memoria assegnatoli in memoria centrale, contenente le sue istruzioni ed i suoi dati, e nei registri della CPU durante la sua esecuzione. Questi rappresentano essenzialmente un’estensione della RAM disponibile al processore. Memorizzano l’indirizzo della prossima istruzione da eseguire, il “Program Counter” (PC), e l’indirizzo dello stack, lo “Stack Pointer” (SP), associato a questa esecuzione del programma, i più importanti.

Il sistema operativo controlla il ciclo di vita dei processi, gestendo una serie di funzioni, disponibili attraverso l’API, “Application Programming Interface” dei processi:

- Create: crea un nuovo programma, caricandolo dal disco ed allocando le risorse necessarie;
- Destroy: interrompe l’esecuzione di un processo, attraverso una GUI, oppure con un comando “kill” o “kill all”. Questo processo non danneggia l’applicazione;
- Wait: mette in pausa l’esecuzione di un processo fino a quando non avviene un evento specifico;
- Miscellaneous Control: offre metodi vari per la sospendere l’esecuzione di un processo per poi riprenderla in seguito. È compreso nei meccanismi di time-sharing per permettere l’esecuzione degli altri processi in coda;
- Status: permette di ottenere informazioni riguardo il processo attuale.

3.2.1 Creazione di un Processo

Per effettuare un’operazione di creazione il sistema operativo effettua una serie di operazioni. Durante l’operazione di “Load” alloca lo spazio in memoria centrale per accomodare il programma. In seguito il programma, memorizzato sulla memoria secondaria in formato eseguibile, e viene trasferito in maniera “lazy”, pigra. Non viene trasferita la sua integrità, ma solo le istruzioni strettamente necessarie all’esecuzione del programma, e vengono trasferite ulteriori istruzioni quando vengono richieste da altre istruzioni nel processo in esecuzione. Nell’operazione di “Stack Allocation” alloca lo stack di esecuzione del programma, utilizzato per memorizzare le variabili locali, parametri di funzioni e indirizzi di ritorno. Inizializzato con gli argomenti passati alla funzione `main` nel programma, come `argc`, numero degli argomenti passati, e l’array degli stessi `argv`.

Nell’operazione di “Heap Creation” viene realizzato lo spazio di heap dove vengono contenuti i dati salvati durante l’esecuzione del programma, tramite operazioni di `malloc()`, richiedendo

esplicitamente lo spazio. Questo viene poi liberato esplicitamente dal programma con un'operazione di `free()`, altrimenti viene liberato al termine della sua esecuzione.

"Initialize": vengono inizializzati i tre descrittori di default per il programma, lo `stdin`, standard input; lo `stdout`, standard output; e lo `stderr`, standard errore. Canali attraverso cui il processo è in grado di comunicare con l'ambiente esterno tramite meccanismi di I/O, oppure notificare il sistema operativo di eventuali errori.

Alla fine di questo meccanismo di creazione, il sistema operativo trasferisce il controllo della CPU al processo, iniziando dalla prima istruzione dopo la funzione `main()`, punto di ingresso in molti linguaggi di programmazione.

3.2.2 Divisione in Stati

Un processo può esser in tre stadi distinti durante la sua esecuzione. Il sistema operativo gestisce la transizione tra questi stati e garantisce che i processi vengano eseguiti correttamente, in modo efficiente.

- **Running**: il processo è in esecuzione, ovvero ha il controllo del processore ed esegue le sue istruzioni;
- **Ready**: il processo è pronto per essere eseguito, ma il sistema operativo non l'ha scelto e quindi aspetta il suo turno nella coda;
- **Blocked**: il processo è bloccato, perché ha richiesto delle operazioni ad alcuni dispositivi connessi al calcolatore. Impedisce che il processore rimanga occupato da un processo che non può effettivamente eseguire istruzioni.

In realtà sono presenti molti più stadi per lo stato di un processo, ma questa rappresenta una buona approssimazione.

Un processo pronto può essere schedulato per essere eseguito, oppure è possibile de-schedulato un processo in esecuzione per renderlo pronto. Un processo in esecuzione quando richiede accesso a memoria o a dispositivi di I/O viene bloccato, e poi sbloccato quando finisce questa operazione, diventando pronto per riprendere la sua esecuzione.

Con questi stati quindi i thread di esecuzione dei vari processi vengono intrecciati tra di loro ed il processore è sempre in esecuzione di un processo in ogni quanto di tempo. Per organizzare i dati necessari a gestire questi processi il sistema operativo utilizza strutture dati apposite, per memorizzare lo stack di esecuzione dei processi.

Si considera una di queste strutture dati:

```
// the registers xw6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip; // Instruction pointer register
    int esp; // Stack pointer register
    int ebx; // Called the base register
    int ecx; // Called the counter register
```

```
    int edx; // Called the data register
    int esi; // Source index register
    int edi; // Destination index register
    int ebp; // Stack base pointer register
};
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem; // Start of process memory
    uint sz; // Size of process memory
    char *kstack; // Bottom of kernel stack
                // for this process
    enum proc_state state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, has been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    struct context context; // Switch here to run process
    struct trapframe *tf; // Trap frame for the
                // current interrupt
};
```

Questo codice mostra come il sistema xv6 traccia e gestisce i vari processi ed i registri per l'esecuzione dei vari processi. L'informazione relativa ai vari processi viene salvata nella struttura di tipo `proc`. Queste informazioni permettono di eseguire, terminare o riprendere l'esecuzione di un processo. Fornisce informazioni sulla memoria utilizzata dal processo, i registri utilizzati, il suo stato, i file aperti e la directory corrente, il puntatore al processore genitore.

3.2.3 User e Kernel Mode

Il sistema operativo deve effettuare queste operazioni di time sharing efficientemente. Quindi bisogna minimizzare il tempo tra la fine di un processo e l'inizio del successivo. In questo modo si minimizza l'overhead. Oltre a risolvere questo problema di overhead, il sistema operativo deve riprendere il controllo dopo che il quanto di tempo assegnato a quel processo esaurisce, altrimenti potrebbe essere eseguito ininterrottamente, senza liberare le sue risorse. Inoltre se non mantiene il controllo del processore, un processo maligno potrebbe effettuare operazioni non autorizzate e mettere a rischio il calcolatore. Il sistema operativo deve essere in grado di poter effettuare controlli di sicurezza per verificare se un dato processo sia autorizzato a compiere una certa azione.

Per risolvere il problema delle prestazioni, il processo non viene eseguito attraverso il sistema operativo, come fosse un tramite, ma viene eseguito direttamente sul processore. Questa tecnica si chiama esecuzione diretta, senza passare un livello intermedio di emulazione o interpretazione. In questo modo si riduce al minimo l'intervento del sistema operativo durante l'esecuzione. Se il sistema operativo non mantenesse il controllo sarebbe ridotto ad una semplice libreria, per svolgere funzioni importanti dei vari processi, senza poter gestire le varie risorse. Ma se il processo viene eseguito direttamente sul processore, allora il sistema operativo non può essere eseguito per mantenere il controllo, e se il processo richiede un'operazione non consentita, il sistema operativo non ha modo di impedirglielo.

La soluzione per risolvere l'esecuzione di operazioni proibite consiste nell'avere due modalità di esecuzioni diverse, una per l'utente e l'altra per il sistema operativo. Nello spazio o modalità utente, i processi non hanno accesso a tutte le risorse hardware del calcolatore, mentre nella modalità kernel, il sistema operativo può utilizzare tutte le risorse della macchina. Le chiamate di sistema permettono ai processi di interfacciarsi con lo spazio kernel del sistema.

Nello spazio utente i processi vengono eseguiti con accesso limitato alle risorse dell'hardware, mentre le operazioni critiche come accessi in memoria, o a dispositivi di I/O non sono permesse direttamente. Nella modalità kernel ha pieno accesso a tutte le risorse del calcolatore, inclusi hardware e memoria. In questa modalità il sistema operativo esegue le operazioni richieste dai processi nello spazio utente, in questo modo gli permette di effettuare controlli di sicurezza su queste operazioni.

In generale per distinguere tra queste due modalità, è presente un flag in un registro del processore per segnalare in quale modalità ci si trova. In questo modo il processore può rifiutare le istruzioni ristrette in base al valore di questo flag. In pratica abilita il set completo delle istruzioni del kernel.

3.2.4 Chiamate di Sistema

Si utilizzano le chiamate di sistema, interfaccia sicura e standardizzata per permettere ai processi di accedere a risorse di sistema o servizi critici. Permettono di eseguire operazioni come accedere al filesystem, creare o distruggere processi, comunicare con altri processi o allocare memoria. Per eseguire una chiamata di sistema il processo invoca una speciale istruzione di "trap", che effettua insieme un salto nel kernel, ed un aumento del livello di privilegio alla modalità kernel. Viene realizzata al livello del processore, come un'operazione di salto alla prima istruzione della chiamata di sistema nel kernel. Il controllo allora passa al sistema operativo, all'interno del kernel, e viene modificato il PC.

Dopo aver eseguito l'operazione richiesta dall'utente, per restituire il controllo al processo, viene invocata una chiamata "return-from-trap", ritorna all'istruzione subito seguente dalla chiamata di sistema nel programma e diminuisce il livello di privilegio allo spazio utente.

Ma bisogna impedire di effettuare salti arbitrari dentro al kernel, questo permetterebbe l'esecuzione di codice arbitrario all'interno del kernel, il quale potrebbe essere molto dannoso in caso sia presente un processo maligno o contenente errori o bug. Per questo al tempo di avvio del calcolatore viene generata una tabella chiamata "trap table" contenente l'indirizzo della prima istruzione di ogni chiamata di sistema, numerandole. In questo modo si può eseguire il codice del kernel in modo

sicuro e solamente in punti predefiniti, riducendo il rischio di un'esecuzione dannosa di istruzioni nel kernel. Queste chiamate di sistema rappresentano l'unica un'interfaccia o API tra il processo nello spazio utente e le istruzioni del nello spazio del kernel. Tramite la trap table. A livello di hardware viene impostato su un registro l'indirizzo del gestore delle chiamate di sistema.

3.2.5 Ciclo di Vita di un Processo

All'avvio di un programma viene creata un'entità processo da parte del sistema operativo, per allocare la memoria, ad una struttura dati del processo di tipo `proc`. Il codice del programma viene poi trasferito in memoria centrale, dal disco, in maniera lazy, ovvero trasferendo solamente il codice strettamente necessario, e successivamente si trasferisce solo il codice che viene utilizzato. Inoltre si realizza lo stack e l'heap dell'applicazione, e si realizza il doppio puntatore a caratteri `argv`, per gestire gli input per il processo. Si inserisce nello stack del kernel il registro ed il PC per tenere conto del processo in esecuzione, e tiene conto delle chiamate di sistema e dei loro argomenti per effettuare system call.

Alla fine di queste operazioni bisogna passare il controllo al programma appena avviato, utilizzando una chiamata "return from trap", e bisogna specificare la prima istruzione da eseguire. L'hardware conosce l'indirizzo di memoria base dello stack che contiene il valore dei registri da inizializzare per avviare il processo. Dopo aver popolato i registri ed il PC, cambia la modalità di funzionamento allo spazio utente e salta alla prima istruzione nel metodo `main`.

Ora il programma in esecuzione segue le istruzioni sequenzialmente, ed eventualmente deve eseguire delle chiamate di sistema, quindi invoca una funzione di trap per saltare al livello spazio kernel, ma bisogna salvare i valori del processo nello spazio riservato nel kernel. Questo salvataggio viene effettuato in modo completamente autonomo da parte dell'hardware prima di cedere il controllo al kernel. Dopo averli salvati si eleva il livello di privilegi e si salta alla prima istruzione del gestore della trap, eseguito dal punto di vista del kernel, si eseguono le istruzioni della chiamata di sistema, e si effettua una istruzione `return from trap`. Il processore rimuove i valori salvati nello stack del kernel e si sposta allo spazio utente. Inoltre salta all'istruzione indicata dal PC dopo la chiamata trap.

Alla terminazione del processo non bisogna salvare i suoi valori poiché non avrà modo di effettuare altre chiamate di sistema, quindi l'hardware non svolge nessuna operazione. Il kernel quindi svuota svuota l'area di memoria assegnata al processo e lo rimuove dalla struttura dati utilizzata per contenere la lista dei processi.

3.2.6 Gestione Efficiente dei Processi

Il sistema operativo ha come obiettivo la ricerca di prestazioni più alte possibili, bisogna quindi evitare di inserire livelli in più tra il processo al processore, per impedire rallentamenti nell'esecuzione del processo. Ma allo stesso tempo si vuole mantenere il controllo del processore da parte del sistema operativo. Data la modalità descritta precedentemente il processo restituisce il controllo al sistema operativo dopo una chiamata di sistema, quindi il processore mentre l'esecuzione del processo non sta eseguendo il sistema operativo. Questo rappresenta quindi un problema, poiché un processo malevolo potrebbe non effettuare mai chiamate di sistema, quindi non restituisce mai il controllo al sistema operativo e viene bloccata l'esecuzione del sistema operativo. Questo può

essere realizzato facilmente tramite un ciclo infinito. Non necessariamente però dipende da una programmazione malevola, potrebbe essere causata ad un errore o bug nel programma.

Quindi è necessario cambiare l'approccio per riguadagnare il controllo del processore dal sistema operativo. Si definiscono quindi due approcci, uno cooperativo, l'altro non cooperativo. Il primo approccio consiste nella modalità precedente, dove il sistema operativo aspetta una chiamata di sistema per riprendere il controllo del processore, oppure se viene sollevata una qualche tipo di eccezione. Esiste una chiamata di sistema `yield()` che permette di cedere il controllo del processore al sistema operativo. Ma non è garantito che ogni processo contenga una chiamata di sistema, e che vengano chiamate periodicamente.

Si vorrebbe utilizzare un sistema dove il controllo viene ceduto al sistema operativo periodicamente, questo tuttavia non può essere realizzato interamente tramite software. Bisogna quindi ricorrere al supporto dell'hardware, generando un timer all'avvio di ogni processo. Quando questo timer arriva al termine viene invocato un interrupt che passa il controllo al sistema operativo, da parte di un gestore di interrupt. Questo rappresenta uno scheduling non cooperativo. I processi che vengono eseguiti non gestiscono il controllo del processore. In questo modo è possibile al sistema operativo riprendere il controllo nonostante la presenza di applicazioni malevoli. Questo timer lavora nell'ordine dei millisecondi, ed è indipendente dalla frequenza di aggiornamento del processore.

Un processo potrebbe non terminare tra due di questi timer interrupt, quindi il controllo passa al sistema operativo alla terminazione del programma, ed in alcuni casi viene disattivato il gestore degli interrupt e si scartano alcuni interrupt. Se dei processi hanno una priorità diversa, invece di terminare la loro esecuzione al primo interrupt, il controllo viene lasciato al processo invece del sistema operativo, fino ad un certo numero di interrupt.

Viene chiamata la funzione scheduling ad ogni interrupt per scegliere se si continua ad eseguire il processo che era in esecuzione oppure eseguire un nuovo processo. Quando viene scelto di cambiare il processo da eseguire, si effettua un context switch, un cambio di contesto, poiché tutti i valori associati al processore e dei registri vengono salvati per usarne di nuovi.

Con questa modalità, all'avvio si inizializza la trap table, ed il gestore degli interrupt decide l'intervallo tra gli interrupt. Quando viene sollevato il timer interrupt all'esecuzione di un processo, l'hardware effettua operazioni analoghe alla chiamata di sistema. I suoi valori e registri vengono salvati nel suo stack di processo, ed il processore cambia la modalità in kernel mode e passa il controllo al sistema operativo. Il sistema operativo sceglie dal suo scheduler a quale processo restituire il controllo del processore. Scelto il processo, l'hardware riassegna i valori ed i registri e salta allo spazio utente.

Se viene sollevato un interrupt mentre il processo sta scrivendo sui dati sul disco, oppure durante l'esecuzione di una chiamata di sistema, il sistema operativo può reagire in modi diversi. Potrebbe scartare gli interrupt sollevati durante una chiamata di sistema, oppure potrebbe usare sofisticati meccanismi di lock per proteggere l'accesso a strutture dati interne. Questi meccanismi saranno analizzati dettagliatamente in una successiva sezione del corso.

3.3 Scheduling

Dopo aver trattato il meccanismo di context-switch bisogna analizzare le politiche di scheduling. Verranno utilizzate una serie di assunzioni irrealistiche. La prima di queste assunzioni è che tutti i "job", lavori che il sistema deve eseguire, abbiano lo stesso tempo di esecuzione. Si assume che tutti i lavori arrivino allo stesso momento. Si considera che ciascuno di essi venga eseguito fino al suo completamento. Ognuno di questi utilizza esclusivamente il processore, non effettuano operazioni di input/output. Si suppone infine che si conosca a priori il tempo di esecuzione di tutti questi job.

Si vuole misurare in modo quantitativo le politiche di scheduling, si introducono quindi una serie di metriche. La prima chiamata turnaround time, è il tempo di lavoro di un job, si calcola come la differenza tra il tempo di arrivo ed il tempo di completamento, data l'assunzione che tutti questi job arrivino allo stesso momento, si impone sia nullo. Quindi con queste assunzioni il tempo di turnaround corrisponde al tempo di completamento. La seconda metrica è la fairness, quanto le risorse vengono divise in modo equo rispetto a tutti i processi.

3.3.1 First Come, First Served

La prima politica di scheduling è la FIFO, o FCFS, "First Come, First Served", semplice e di facile implementazione. Si suppone arrivino sequenzialmente a distanza temporale minima, il primo job ad arrivare sarà il primo ad essere eseguito.

3.3.2 Shortest Job First

Nel momento in cui si considerano situazioni più realistiche, questa politica perde di efficienza, poiché il primo job potrebbe richiedere un tempo molto superiore ai successivi, quindi si verifica l'effetto convoglio. Il primo "vagone" è molto più grosso dei seguenti e quindi il turnaround time è molto elevato.

Nello stesso contesto si può utilizzare una gestione diversa, per migliorare il tempo di ritorno. Si considera allora una politica che esegue i lavori più corti prima, SJF "Shortest Job First", noto il tempo di esecuzione. In questo modo si minimizza il turnaround time, decisamente più basso della politica FIFO.

In uno scenario più realistico, non tutti i job arrivano nello stesso momento nel sistema, quindi è possibile che il primo lavoro che arriva sia l'unico possibile da eseguire, e sia anche il più lungo, quindi in queste condizioni si verifica nuovamente l'effetto convoglio.

3.3.3 Shortest Time to Completion First

Per cui si considera un sistema dove se arriva un job che richiede un tempo minore per essere completato, viene interrotta l'esecuzione del processo attuale e viene eseguito questo nuovo processo. Questa politica si chiama STCF "Shortest Time to Completion First", aggiunge un meccanismo di prevenzione a SJF, quindi viene chiamato anche PSJF, "Preemptive Shortest Job First". Ogni volta che un job entra nel sistema ne calcola il tempo di completamento e se è minore del processo attualmente in esecuzione allora si interrompe l'esecuzione del processo corrente e si esegue il nuovo processo. Quindi all'arrivo di un nuovo processo, il sistema operativo deve determinare il tempo di completamento per questi nuovi processi e per il processo attualmente in esecuzione.

3.3.4 Round Robin

Esistono altre metriche altrettanto significative come il tempo di risposta, definito come il tempo in cui il job è stato in coda, prima di essere eseguito per la prima volta. Un modo per minimizzare o controllare il tempo di risposta per un determinato job si considera la politica di scheduling "Round Robin" (RR). Questa politica effettua una suddivisione nel tempo cambiando periodicamente quale dei processi in coda viene eseguito, fino all'esaurimento dei job in coda. In questo modo viene garantito a tutti i job un quanto di tempo, essenzialmente l'intervallo di tempo tra due interrupt. Ma è possibile che questo quanto di tempo sia un multiplo del timer interrupt, in modo per sintonizzare gli interrupt e questi quanti di tempo. Secondo la definizione di fairness, questo è una politica fair poiché assegna ad ogni job la stessa quantità di CPU. Ma ottiene delle prestazioni pessime rispetto al tempo di ritorno. Infatti per minimizzare il tempo di ritorno il processore deve concentrarsi su un unico job alla volta.

La lunghezza del quanto di tempo è critico per il tempo di risposta. Diminuendo il quanto di tempo, l'overhead rappresentato dal context switch rappresenta una parte considerevole del quanto di tempo, ma diminuisce il tempo di risposta. Se aumenta il quanto di tempo, viene ammortizzato l'effetto del context switch, il tempo di risposta aumenta. Questa politica è tra le peggiori per il tempo di ritorno, ma è la più equa nella sua distribuzione delle risorse.

Si considerano ora job in grado di eseguire operazioni input/output o accessi a risorse esterne al processore, quindi possono passare dallo stato running allo stato blocked.

3.3.5 Multi-Level Feedback Queue

In questi algoritmi il turnaround time ed il tempo di risposta sono inversamente proporzionali tra di loro, e non è stato possibile trovare un bilanciamento tra di loro. Queste due famiglie di algoritmi che minimizzano queste due caratteristiche sono separati fra di loro, e supportano due tipologie di programmi separate. Si utilizzano politiche di tipo round robin per programmi interattivi, dove si vuole minimizzare il tempo di attesa con l'utente stesso. Invece per i sistemi batch si vuole minimizzare il tempo di esecuzione complessivo del sistema, e si massimizza il throughput dei job realizzabili. L'algoritmo "Multi-Level Feedback Queue" MLFQ, si ottiene rilassando il vincolo sulla conoscenza a priori la durata di un job. Questo algoritmo prova ad minimizzare entrambi questi parametri. Questo algoritmo ha una serie di regole basi. Si assume che esistano una serie di code, gestite in base a livelli di priorità, quindi non permette all'utente di variare il livello di priorità di un job. Rappresenta un meccanismo interno utilizzato per inseguire gli obiettivi dell'algoritmo.

Se è presente un job nella coda a priorità più alta, sicuramente verrà eseguito. Queste varie code contengono job di stessa priorità, questi job di pari priorità verranno eseguiti in modalità RR. In base a queste due regole l'algoritmo è in grado di minimizzare queste due parametri caratteristici. Inoltre utilizza dei meccanismi di analisi del comportamento dei job per poter determinare dinamicamente la loro priorità. Se un job continuamente rilascia la CPU, per continue operazioni di I/O si vuole mantenere la sua priorità alta, si tratta quindi di un job interattivo, e si vuole ridurre il più possibile il suo tempo di risposta. Un job che non rilascia mai la CPU invece viene associato ad una priorità minore, poiché si comporta come un job di tipo batch. Questi tipi di job verranno messi nella coda più bassa, quindi si aumenta il loro turnaround time, ma poiché ha un tempo di completamento

molto elevato, il turnaround time non rappresenta un overhead significativo. Una terza regola impone che ogni job al primo arrivo nel sistema sia trattato come un job interattivo.

Per la quarta regola, se un job impiega l'intero quanto di tempo assegnatoli, allora diminuisce la sua priorità. Se un job rilascia la CPU prima del termine dell'intervallo di tempo concessogli, allora la sua priorità rimane invariata.

Con l'insieme di regole definite, questo algoritmo approssima l'algoritmo STCF, ma allo stesso tempo non assume di conoscere la lunghezza di un job. Se un job di priorità elevata, e quindi di durata, probabilmente, minore, arriva nel sistema; questo interrompe l'esecuzione del job di priorità più bassa e durata maggiore per eseguire il job di priorità più alta. Questo emula il comportamento di STCF.

Questo algoritmo tuttavia non è perfetto. Se sono presenti abbastanza job interattivi per saturare la CPU, allora i nelle code di priorità di livello inferiore potrebbero andare in "starvation", ovvero le risorse del sistema non verranno mai assegnate a loro. Inoltre è possibile ingannare lo scheduler, un programma malevolo potrebbe eseguire una serie di operazioni ed in seguito richiedere un'operazione di I/O, riceverà più operazioni di quanto un sistema a regime potrebbe. In questo modo si potrebbe usare l'intervallo di tempo assegnato per la maggior parte, ed in seguito rilasciare la CPU poco prima dello scadere. Un programma invece potrebbe cambiare il suo comportamento nel tempo, cambiando da un job batch ed interattivo.

Per provare a risolvere questi problemi, si introduce una nuova regola. Dopo un certo intervallo di tempo tutti i job nel sistema vengono inseriti nella coda più alta. Questo "Priority Boost" consente ai programmi in rischio di starvation di avere una possibilità nell'essere eseguiti. Allo stesso modo per i programmi in cui cambiano di comportamento, infatti potrebbero essere elevati di priorità al loro cambiamento.

Non è possibile conoscere in termini generali il tempo di attesa per effettuare questo priority boost, cambia da sistema a sistema, e rappresenta uno dei parametri più importanti di questo algoritmo.

Un sistema antivirus per effettuare controlli degli altri job, richiederebbe di un livello di priorità ancora superiore. Per risolvere questo gli antivirus vengono inseriti in una parte specifica del kernel, in modo che possono effettuare i loro controlli al tempo di context switch.

Invece per evitare che lo scheduler venga ingannato viene modificata la quarta regola dell'algoritmo. In questo modo impone che tutti i job che hanno utilizzato complessivamente tutto il loro quanto di tempo, avranno la loro priorità ridotta. Non solo quindi se esaurisce il quanto di tempo a sua disposizione.

La maggior parte degli algoritmi MLFQ utilizzano quanti di tempo elevati per code di bassa priorità, nell'ordine dei 100 ms, e tempi nell'ordine dei 10 ms per code di priorità più elevata.

Il sistema UNIX Solaris implementa l'algoritmo MLFQ utilizzando 60 code diverse, con un aumento graduale dell'intervallo di tempo assegnato ad ogni job, dai 20 alle centinaia di millisecondi. Ed effettua un priorità boost ogni secondo circa.

3.3.6 Completely Fair Scheduling

Circa il 5% del tempo di CPU di tutti i data center Google sia dedicato allo scheduling, quindi è di principale importanza definire algoritmi di scheduling efficienti per migliorare le prestazioni.

Il sistema Linux utilizza un algoritmo diverso per effettuare il suo scheduling, chiamato "Completely Fair Scheduling" (CFS), lo scheduler attuale di tutte le macchine Linux. Le sue principali caratteristiche sono l'assenza di un quanto di tempo fisso, il supporto delle priorità, modificabili da parte dell'utente. Questo controllo viene gestito tramite un parametro chiamato "nice", per definire quanto un processo debba essere gentile rispetto ad altri processi. I valori di nice negativi indicano una priorità elevata. Utilizza inoltre delle strutture dati molto efficienti per la ricerca, l'inserimento e l'eliminazione dei processi.

Il parametro "virtual runtime", *vruntime*, rappresenta per quanto tempo il processo è stato eseguito dal momento in cui è stato avviato. Viene incrementato proporzionalmente rispetto al tempo reale di utilizzo della CPU. In generale l'algoritmo CFS seleziona sempre il processo di *vruntime* più basso.

Il parametro *sched_latency* indica l'intervallo di tempo dove viene eseguita la suddivisione in quanti temporali. Definendo questo parametro, è possibile garantire vincoli impliciti al tempo di risposta. Un valore tipico è di 48 ms. La durata di un quanto di tempo corrisponde a questo valore diviso per il numero di processi nel sistema.

Il parametro *min_granularity* indica l'intervallo minimo per un quanto di tempo, in modo da garantire che l'overhead del tempo di scheduling non sia eccessivo. Quando si arriva a questo tempo viene incrementato lo *sched_latency*. Ne risente il tempo di risposta del sistema, ed è indecidibile. Il valore di *nice* varia tra -20 a +19. Questi valori vengono implicitamente tradotti con dei pesi a dei livelli di priorità.

Invece di assegnare in modo completamente equo il quanto di tempo, l'intervallo di tempo destinato ad un singolo processo non è lo stesso. Dato un processo k , il suo intervallo di tempo si ottiene dalla formula:

$$\text{time_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched_latency} \quad (3.3.1)$$

Il quanto di tempo associato ad un processo k aumenta all'aumentare della suo contributo al peso totale di tutti i processi. In presenza di processi di priorità più alta allora bisogna aumentare il loro *vruntime* di una quantità minore rispetto agli altri processi. Questo *vruntime* equivale al tempo di esecuzione bilanciato dal peso:

$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i \quad (3.3.2)$$

Questo algoritmo utilizza un albero rosso-nero per memorizzare i processi in coda. Questo è un albero binario, automaticamente bilanciato ad ogni inserimento o rimozione di un nodo. Vengono mantenuti solo i processi di stato *running* o *ready*, i processi in stato di *blocked* vengono rimossi da questo albero. Permette di individuare in tempo efficiente il processo con il *vruntime* minore. Se un processo vuole effettuare un'operazione di I/O, il suo *vruntime* rimane costante mentre il valore degli altri processi in esecuzione aumenta.