This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Scheduling

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

1. Introduzione e meccanismi di scheduling
2. Metriche e politiche di scheduling
3. Esempi di algoritmi "real-world"
   - Multi-level Feedback Queue
   - Proportional share e Linux CFS (Completely Fair Scheduler)

Introduzione

Per virtualizzare la CPU si utilizza il time sharing; ovvero, ogni processo riceve l'uso (esclusivo) della CPU per un po', poi si passa a un altro

- Il meccanismo che permette di cambiare processo, è chiamato cambio di contesto (context switch)
- Lo scheduler è quella parte di kernel che decide chi è il prossimo processo da mandare in esecuzione, secondo una certa politica (policy)

Problemi:

- Come implementare il context-switch in modo efficiente
- Controllo: come garantire che un processo rilasci l'uso della CPU?

Vedere:
`http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf`

Meccanismi per lo scambio di processi

Approccio

- cooperativo: il SO si "fida" dei processi
  - Ogni tanto i processi eseguono una syscall, anche se non ne hanno bisogno (tipicamente syscall ad-hoc: `yield`)
  - Cosa succede se non lo fanno, per malizia o per un bug?
- non cooperativo: il SO si riprende il controllo "a forza", tramite un timer interrupt

Context-switch

L'idea è semplice:

1. si salvano i registri del processo che si sta eseguendo
2. si caricano i registri del processo che vogliamo mandare in esecuzione
    - nota: questo include anche, per esempio, il registro che "punta" alla tabella delle pagine

. . . l'implementazione un po' meno: molti dettagli dipendenti da HW

Context-switch e timer-interrupt

In generale, ogni processo ha uno user stack e un kernel stack:

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

da: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf`

Stati di un processo

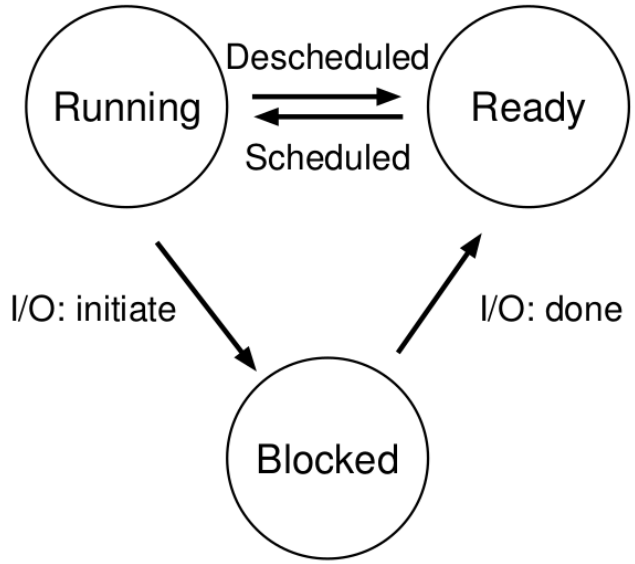


Figure 4.2: **Process: State Transitions**

da: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf`

Per esempio, su Xv6:

```
enum procstate { UNUSED, EMBRYO /* AKA: INIT */, SLEEPING,
                 RUNNABLE /* AKA: READY */, RUNNING, ZOMBIE };
```

In altri Unix-like, es. Linux, le cose possono essere più complicate (due diversi tipi di *sleep*, anche stato uno *stopped*, . . .)

Outline

1. Introduzione e meccanismi di scheduling
2. Metriche e politiche di scheduling
3. Esempi di algoritmi "real-world"
   - Multi-level Feedback Queue
   - Proportional share e Linux CFS (Completely Fair Scheduler)

Valutazione degli algoritmi di scheduling

Questa parte è basata su:
`http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf`

- i processi che girano in un sistema sono detti *workload*
- nel contesto dello scheduling i processi sono spesso chiamati *job*

Assunzioni

Partiamo con assunzioni *irrealistiche*

1. ciascun job dura lo stesso tempo
2. tutti i job arrivano allo stesso momento
3. una volta iniziato un job lo si porta fino in fondo (senza interruzioni)
4. tutti i job usano solo CPU, no I/O
5. il tempo di ciascun job è noto a priori

e le elimineremo man mano

Metriche

Per misurare la "bontà" di un algoritmo di scheduling, rispetto a un altro, abbiamo bisogno di metriche

- inizialmente ci concentreremo sulle *performance*
- ma un altro aspetto da tenere in considerazione è la *fairness*

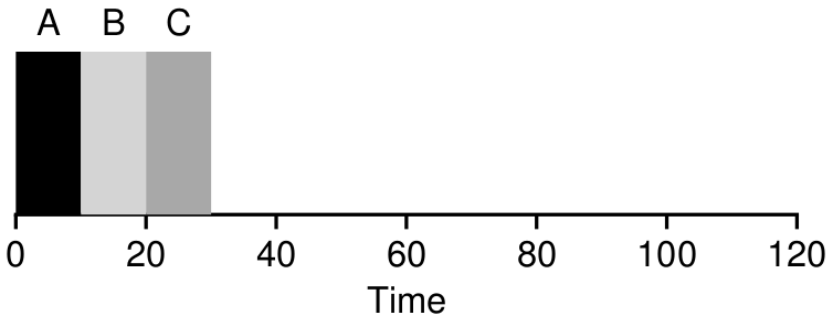
Turn-around time

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

con le assunzioni iniziali, $T_{\text{arrival}} = 0$ quindi:

$$T_{\text{turnaround}} = T_{\text{completion}}$$

Algoritmo FIFO

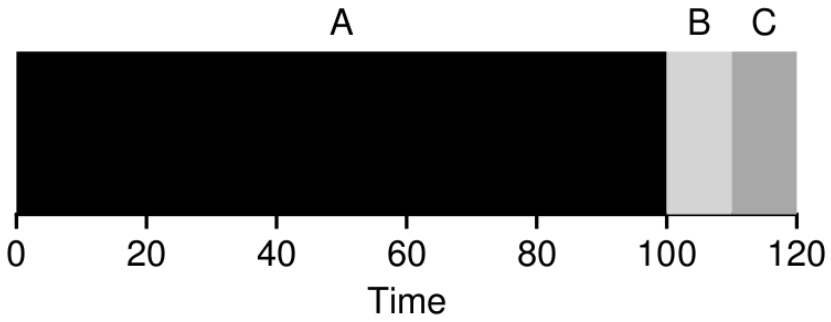


$$T_{\text{turnaround}} = \frac{10 + 20 + 30}{3} = 20$$

Rilassiamo la prima assunzione, cioè che tutti i job durino lo stesso tempo: ci sono workload più brutti di altri?

Effetto convoglio

Se A dura 100, B e C 10

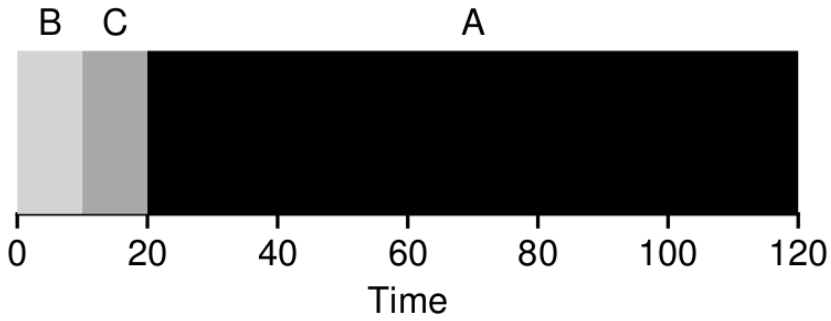


Il tempo medio diventa:

$$T_{\text{turnaround}} = \frac{100 + 110 + 120}{3} = 110$$

per via dello effetto convoglio

Shortest-Job First (1/2)

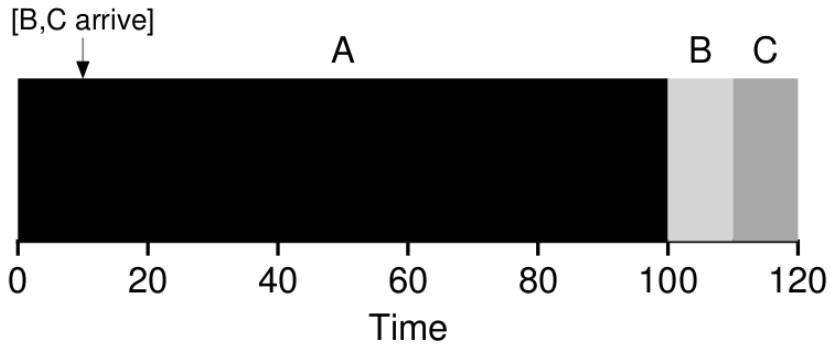


$$T_{\text{turnaround}} = \frac{10 + 20 + 120}{3} = 50$$

Con l'assunzione che tutti i job arrivano allo stesso momento, si può provare che SJF è ottimo. Funziona bene anche se togliamo questa assunzione?

Shortest-Job First (2/2)

Se A arriva all'istante 0, B e C arrivano all'istante 10:

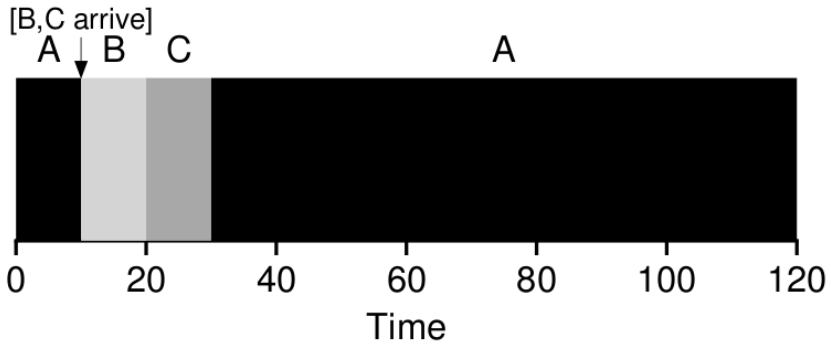


$$T_{\text{turnaround}} = \frac{100 + (110 - 10) + (120 - 10)}{3} \approx 103$$

Cosa si potrebbe fare?

Shortest Time-To-Completion First

Rilassando l'assunzione di non interrompere i job, l'algoritmo STCF è ottimo



$$T_{\text{turnaround}} = \frac{120 + (20 - 10) + (30 - 10)}{3} = 50$$

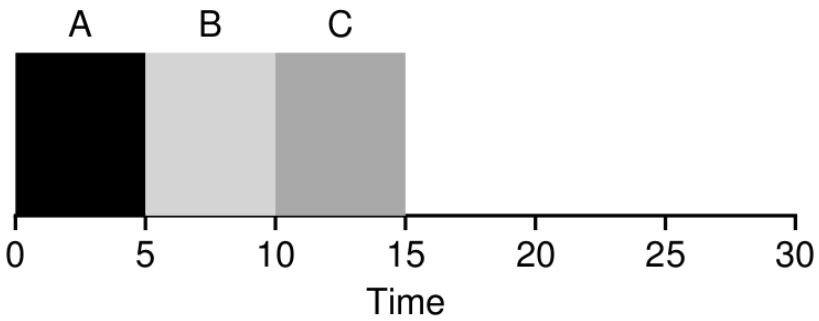
Response time

Nei sistemi batch il turnaround può bastare, ma nei sistemi interattivi un'altra metrica è molto importante

Response time

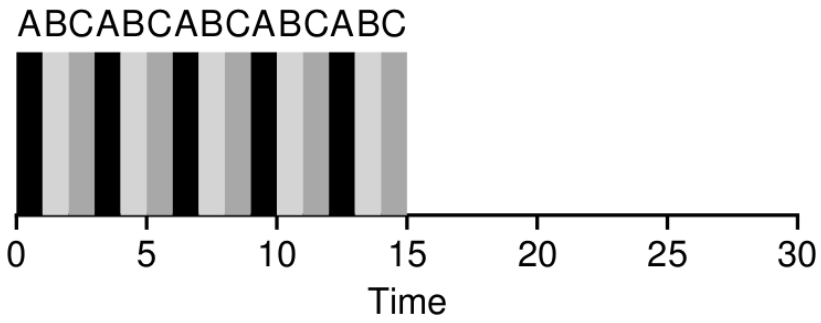
$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$

negli algoritmi visti, il response time non è buono; pensate a cosa succede se arrivano n job della stessa lunghezza: l'ultimo deve aspettare la terminazione degli altri $(n-1)$ prima di essere mandato in esecuzione



Round-Robin (1/2)

Nel Round-Robin ogni job ottiene una "fetta/quanto di tempo" e poi si passa al prossimo job:



Supponiamo che A, B e C arrivino al tempo 0 e durino 5 secondi; il tempo medio con RR sarebbe:

$$T_{\text{response}} = \frac{0+1+2}{3} = 1$$

mentre con SJF/STCF:

$$T_{\text{response}} = \frac{0+5+10}{3} = 5$$

Round-Robin (2/2)

- Si può ottimizzare il response-time restringendo il quanto di tempo, ma bisogna tenere in considerazione l'overhead del context-switch
- Come si comporta RR rispetto al turnaround? Malissimo! L'esecuzione viene "diluita" e il tempo di completamento si allunga
  - in generale gli algoritmi di scheduling "equi" (*fair*) evitano la starvation ma "penalizzano" i job corti

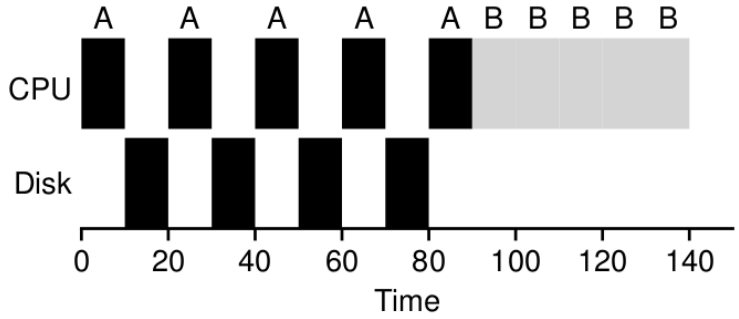
Assunzioni

Abbiamo ancora due assunzioni da eliminare:

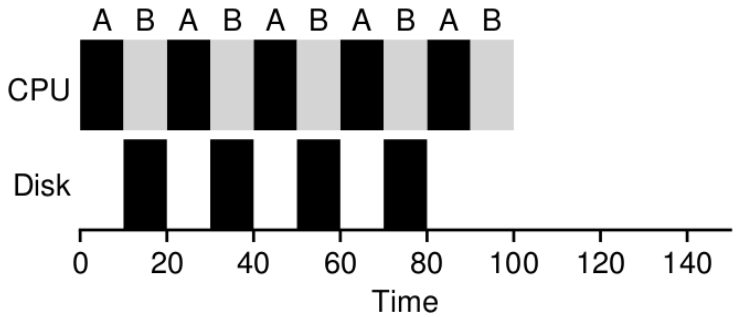
4. tutti i job usano solo CPU, no I/O
5. il tempo di ciascun job è noto a priori

I/O

Come abbiamo, già discusso, sarebbe stupido tenere allocata la CPU per un processo che attende l'I/O:



quindi:



Multi-level Feedback Queue

Il MLFQ tenta di ottimizzare sia il

- turnaround time, facendo eseguire prima i job corti
- response time

In genere uno scheduler non sa a priori le caratteristiche di un processo, come fa, quindi, a "impararle"?

Vedere:
`http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf`

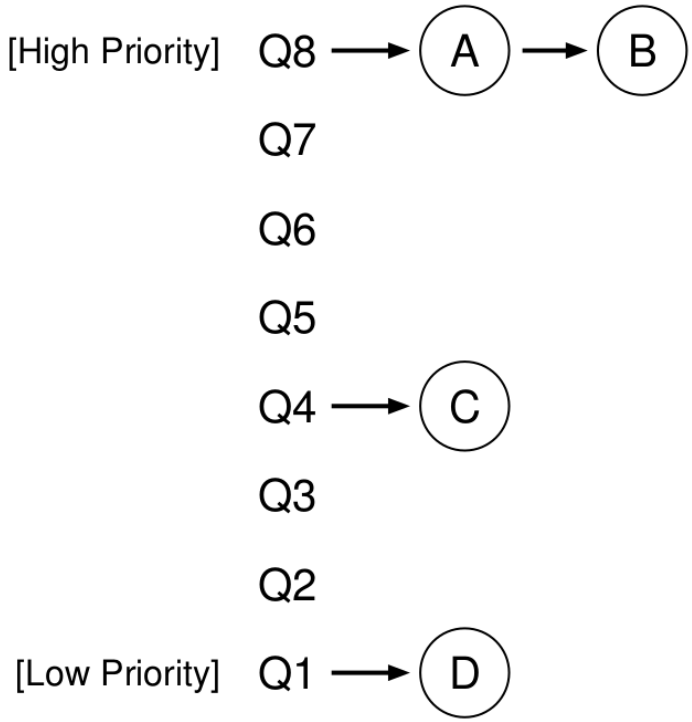
MLFQ: idee

- Diverse code, a differente priorità
  - RR sulla stessa coda
- La priorità di ogni processo varia dinamicamente in base al comportamento osservato
  - CPU vs I/O bound
- Idea: usare la storia di un processo per predirne il futuro

MLFQ: regole

Descriviamo l'algoritmo con delle regole, partiamo da:

1. Se $p(A) > p(B)$, gira A (e non gira B)
2. Se $p(A) = p(B)$, A e B in RR



con solo le prime due regole, *starvation* per C e D

MLFQ: nuove regole — primo tentativo

1. Se $p(A) > p(B)$, gira A (e non gira B)
2. Se $p(A) = p(B)$, A e B in RR
3. Un nuovo job entra con la priorità massima
4. Se un job usa tutto il suo quanto di tempo, allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità)

Funziona?

Problemi

1. starvation: se continuano ad arrivare job, quelli di bassa priorità non verranno mai eseguiti
   - una volta che si è perso priorità, non la si guadagna più (anche se si usa pochissima CPU)
2. si può "barare": se un processo rilascia la CPU al 99% del suo quanto, non perde priorità

MLFQ: nuove regole — secondo tentativo

1. Se $p(A) > p(B)$, gira A (e non gira B)
2. Se $p(A) = p(B)$, A e B in RR
3. Un nuovo job entra con la priorità massima
4. Se un job usa tutto il suo quanto di tempo, allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità)
5. Ogni s secondi, spostiamo tutti i job alla priorità più alta

Funziona? Non c'è più starvation e se un processo diventa I/O-bound, rimane ad alta priorità. Però... si può sempre "barare"

MLFQ: regole finali

1. Se $p(A) > p(B)$, gira A (e non gira B)
2. Se $p(A) = p(B)$, A e B in RR
3. Un nuovo job entra con la priorità massima
4. Quando un job usa un tempo fissato t a una certa priorità x (considerando la somma dei tempi usati nella coda x), allora la sua priorità viene ridotta
5. Ogni s secondi, spostiamo tutti i job alla priorità più alta

Con queste regole le cose più o meno funzionano; problemi:

- Quante code? Quanto vale t? Ogni quanto si resetta tutto?
- La "soluzione" può essere lasciar scegliere i parametri a un admin

Proportional share

Gli scheduler proportional-share invece di cercare di ottimizzare i tempi di workaround o response si basano su un concetto semplice: ogni processo dovrebbe avere la sua percentuale di tempo di CPU

Vedere:
`http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf`

CFS

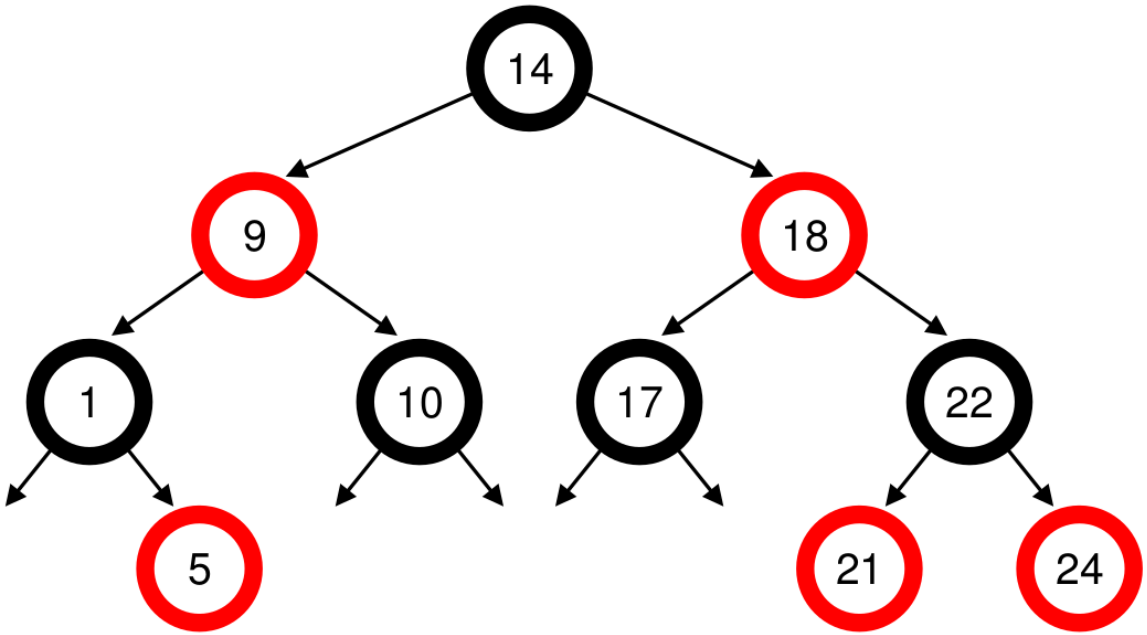
- CFS conteggia il tempo usando virtual runtime: `vruntime`
- nel caso più semplice, il vruntime è proporzionale al tempo vero
- quando c'è da scegliere un processo da mandare in esecuzione, CFS seleziona quello con il `vruntime` più piccolo
  - per quanto? C'è da considerare l'overhead del context-switch
- CFS usa vari parametri, uno è `sched_latency`
- se ci sono `n` processi pronti, si manda in esecuzione quello col `vruntime` più basso per un tempo `sched_latency/n`
  - purché non sia inferiore a un altro parametro: `min_granularity`
- si applica inoltre un fattore di scala, per tenere conto nel valore `nice` (priorità) di ogni processo, per cui:

$$\texttt{slice}_k = \max(\texttt{sched\_latency} \cdot \frac{w_k}{\sum_{i=0}^{n-1} w_i}, \texttt{min\_granularity})$$

$$\texttt{vruntime}_k \mathrel{+}= \frac{\texttt{runtime}_k}{w_k}$$

Efficienza: alberi rossi/neri

Per trovare efficientemente il minimo/aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero:



CFS: I/O

Cosa succede quando un processo viene messo in attesa?

- Se quando un processo torna ready lasciassimo inalterato il vruntime, potrebbe monopolizzare a lungo la CPU
- Quindi, un processo che diventa ready (dopo sleep o init) si prende un vruntime uguale al minimo degli altri
  - quindi non si è "completely fair" con i processi che fanno frequentemente I/O