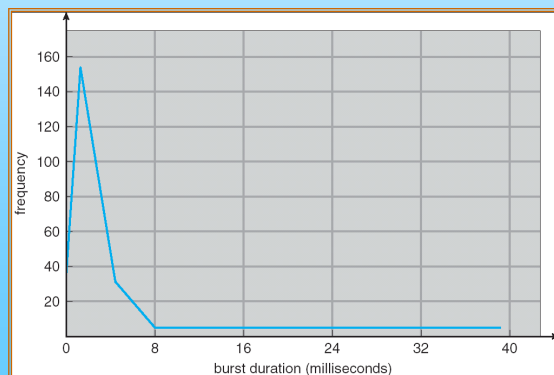
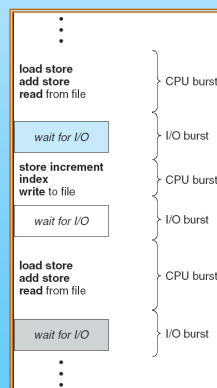


Scheduling della CPU

Concetti base

- L'utilizzazione massima della CPU è ottenuta con la multiprogrammazione
- CPU-I/O Burst Cycle – L'esecuzione dei processi consiste di un ciclo di esecuzione in CPU e attesa di I/O
- Distribuzione della lunghezza dei CPU burst



CPU Scheduler

- Seleziona tra i processi in memoria che sono pronti per essere eseguiti e alloca la CPU a uno di questi (scheduling di breve termine)
- Le decisioni di scheduling della CPU su un processo possono avvenire quando questo passa:
 1. da in esecuzione in attesa
 2. da in esecuzione a pronto
 3. da in attesa a pronto
 4. terminazione
- Lo scheduling che avviene solo nei casi 1 e 4 è detto *nonpreemptive (senza prelazione) o cooperativo*
- Se anche in casi 2 e 3 lo scheduling e' *preemptive (con prelazione)*

Sistemi Operativi A.A. 2016/2017

Prelazione

- Lo scheduling con prelazione può portare a problemi di inconsistenza dei **dati condivisi** tra due processi/thread (**race conditions**)
- Es. un thread viene interrotto durante l'incremento di una variabile in memoria
- Si devono usare meccanismi di sincronizzazione
- Ma quando il processo è sottoposto a prelazione il codice in esecuzione può essere quello del kernel (chiamata di sistema) quindi anche lo stesso kernel deve proteggere le proprie strutture dati (es. code attesa devices) da un uso concorrente.
- In alcuni sistemi operativi questo problema viene risolto facendo in modo di attendere il completamento della chiamata di sistema prima del cambio contesto

Sistemi Operativi A.A. 2016/2017

Dispatcher

- Il modulo Dispatcher del sistema operativo dà il controllo della CPU al processo selezionato dallo scheduler; si occupa di:
 - Cambiare contesto (context switch)
 - Passare a modalità utente
 - Saltare nella locazione opportuna nel programma utente per riprendere l'esecuzione
- *Latenza di dispatch* – tempo impiegato dal dispatcher per fermare un processo e far partire un altro.

Sistemi Operativi A.A. 2016/2017

Criteri di Scheduling

- Utilizzazione della CPU – tenere la CPU più occupata possibile
- **Produttività** – numero di processi completati nell'unità di tempo (*throughput*)
- **Tempo di completamento** o **tempo di ritorno** – tempo necessario a eseguire un processo specifico (*turnaround time*)
- **Tempo di attesa** – tempo speso dal processo nella coda di attesa dei processi pronti
- **Tempo di risposta** – tempo impiegato da quando un processo è sottomesso a quando viene eseguita prima istruzione

Sistemi Operativi A.A. 2016/2017

Criteri di Ottimizzazione

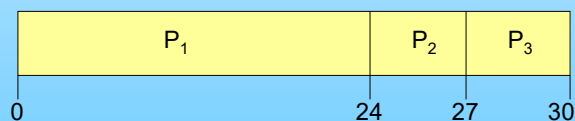
- Massima utilizzazione della CPU
- Massima produttività
- Minimo tempo di completamento
- **Minimo tempo di attesa**
- Minimo tempo di risposta
- Si ottimizzano i valori medi oppure i valori minimi/massimi
- Lo scheduling impatta principalmente sul tempo di attesa

Sistemi Operativi A.A 2016/2017

Scheduling First-Come, First-Served (FCFS)

Processo	T. esecuz.	T.arrivo
P_1	24	0
P_2	3	0
P_3	3	0

- **Senza prelazione**
- Supponiamo che i processi arrivino nell'ordine: P_1 , P_2 , P_3
Il **diagramma di Gantt** per la schedula è:



- **Tempo di attesa** per $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- **Tempo medio di attesa:** $(0 + 24 + 27)/3 = 17$
- **Tempo medio di completamento:** $(24 + 27 + 30)/3 = 27$
- **Tempo medio di risposta:** $(0 + 24 + 27)/3 = 17$

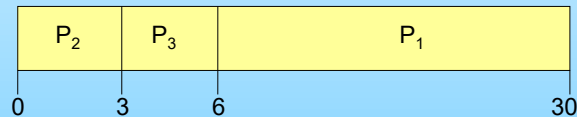
Sistemi Operativi A.A 2016/2017

Scheduling FCFS (Cont.)

Supponiamo che i processi arrivino nell'ordine:

P_2, P_3, P_1

- Il diagramma di Gantt per la schedule è:

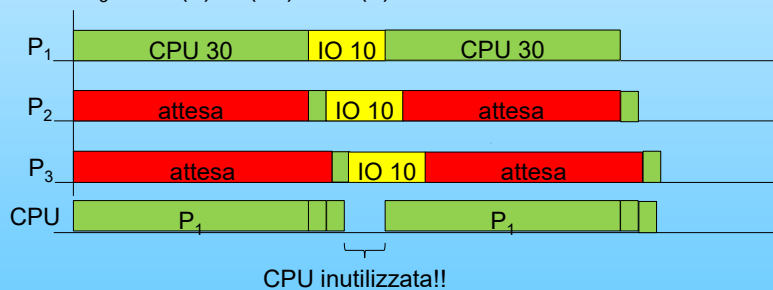


- Tempo di attesa per $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Tempo medio di attesa: $(6 + 0 + 3)/3 = 3$
- Tempo medio di completamento: $(3 + 6 + 30)/3 = 13$
- Tempo medio di risposta: $(0 + 3 + 6)/3 = 3$
- Molto meglio del caso precedente
- *Effetto convoglio* processi corti dietro a processi lunghi

Sistemi Operativi A.A. 2016/2017

Effetto convoglio

- Tre processi:
- P_1 : CPU(30) IO(10) CPU(30) arrivo: 0
- P_2 : CPU(2) IO(10) CPU(2) arrivo: 0
- P_3 : CPU(2) IO(10) CPU(2) arrivo: 0

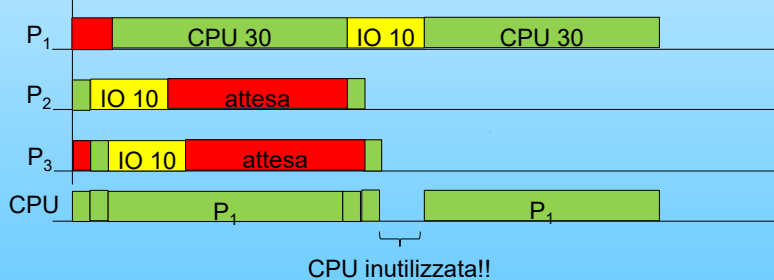


- Tempo attesa medio: $(0 + 58 + 60)/3 = 39,33$

Sistemi Operativi A.A. 2016/2017

Effetto convoglio

- Stessi tre processi eseguiti con ordine: $P_2 P_3 P_1$
- P_1 : CPU(30) IO(10) CPU(30) arrivo: 0
- P_2 : CPU(2) IO(10) CPU(2) arrivo: 0
- P_3 : CPU(2) IO(10) CPU(2) arrivo: 0



- Tempo attesa medio: $(4 + 22 + 24)/3 = 16,66$

Sistemi Operativi A.A 2016/2017

Scheduling Shortest-Job-First (SJF)

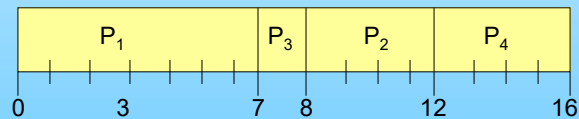
- Associa a ogni processo la **lunghezza del prossimo CPU burst**. Usa queste lunghezze per schedulare il processo con il minor tempo
- Due schemi:
 - **nonpreemptive** – quando la CPU è data a un processo non gli può essere tolta fino alla terminazione del CPU burst
 - **preemptive** – se arriva un nuovo processo la cui lunghezza del CPU burst è minore del tempo rimanente al processo in esecuzione, questo nuovo processo entra in esecuzione. Questo schema è conosciuto come Shortest-Remaining-Time-First (SRTF)
- **SJF è ottimale** – produce il minimo tempo medio di attesa per un insieme di processi fissati

Sistemi Operativi A.A 2016/2017

Esempio di SJF Non-Preemptive

Processo	Tempo di arrivo	Tempo di Burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



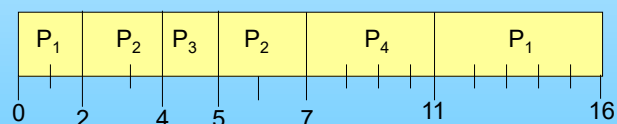
- Tempo di attesa medio = $(0 + 6 + 3 + 7)/4 = 4$

Sistemi Operativi A.A 2016/2017

Esempio di SJF Preemptive

Processo	Tempo di arrivo	Tempo di Burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Tempo di attesa medio = $(9 + 1 + 0 + 2)/4 = 3$

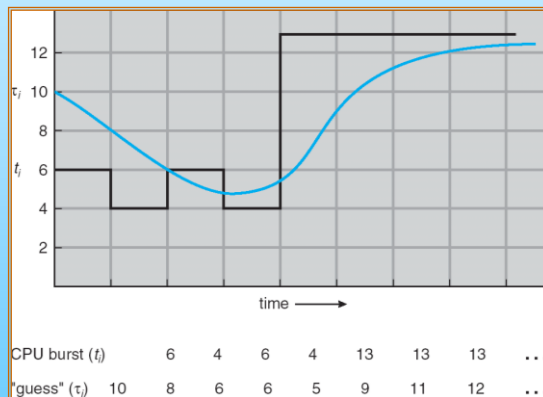
Sistemi Operativi A.A 2016/2017

Determinare la lunghezza del prossimo CPU Burst

- Si può solo stimare la lunghezza
- Può essere fatta usando la lunghezza dei burst precedenti facendo una media esponenziale
 1. t_n = lunghezza dell' n^{esimo} CPU burst
 2. τ_{n+1} = valore predetto per il prossimo CPU burst
 3. τ_0 = valore iniziale preimpostato
 4. $\alpha, 0 \leq \alpha \leq 1$
 5. Si definisce: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

Sistemi Operativi A.A 2016/2017

Predizione della lunghezza del prossimo CPU Burst



Sistemi Operativi A.A 2016/2017

Esempi di media esponenziale

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - La storia recente non conta
- $\alpha = 1$
 - $\tau_{n+1} = t_n$
 - Conta solo l'ultimo CPU burst
- Se si espande la formula otteniamo:
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots$$
$$+ (1 - \alpha)^j \alpha t_{n-j} + \dots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$
- Dal momento che sia α che $(1 - \alpha)$ sono minori o uguali a 1, ogni termine successivo ha minor peso del suo predecessore

Sistemi Operativi A.A 2016/2017

Scheduling con Priorità

- Una priorità (numero intero) viene associato a ciascun processo
- La CPU e' allocata al processo con la priorità più alta (intero più basso \equiv priorità più alta)
 - Preemptive
 - Nonpreemptive
- SJF è uno scheduling a priorità dove la priorità è il tempo predetto del prossimo CPU burst
- Problema: **Starvation** – processi a bassa priorità possono non essere mai eseguiti
- Soluzione: **Aging** (invecchiamento) – si incrementa la priorità al passare del tempo

Sistemi Operativi A.A 2016/2017

Esempio

- Cinque processi arrivati tutti a tempo 0:

Processo	CPU	Priorità
P ₁	10	3
P ₂	1	1
P ₃	2	4
P ₄	1	5
P ₅	5	2



- **Tempo medio d'attesa:** $(6 + 0 + 16 + 18 + 1)/5 = 8,2$
- Usando SJF quale è il tempo medio di attesa?

Sistemi Operativi A.A 2016/2017

Scheduling Round Robin (RR)

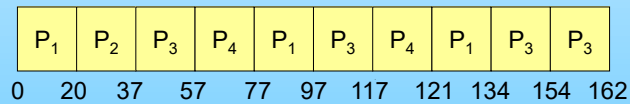
- Ogni processo prende una piccola unità di tempo di CPU (*time quantum*), solitamente 10-100 millisecondi. Dopo che questo tempo è passato il processo viene interrotto e aggiunto alla fine dei processi pronti.
- Se ci sono n processi nella coda dei processi pronti e il quanto di tempo è q , allora ogni processo prende $1/n$ del tempo della CPU in parti di al massimo q unità di tempo alla volta. Nessun processo aspetta più di $(n-1)q$ unità di tempo.
- Performance
 - q grande \Rightarrow FCFS
 - q piccolo $\Rightarrow q$ deve essere grande rispetto al context switch, altrimenti l'overhead è troppo elevato

Sistemi Operativi A.A 2016/2017

Esempio di RR con $q = 20$

Processo	Tempo di Burst
P_1	53
P_2	17
P_3	68
P_4	24

- Il diagramma di Gantt è:



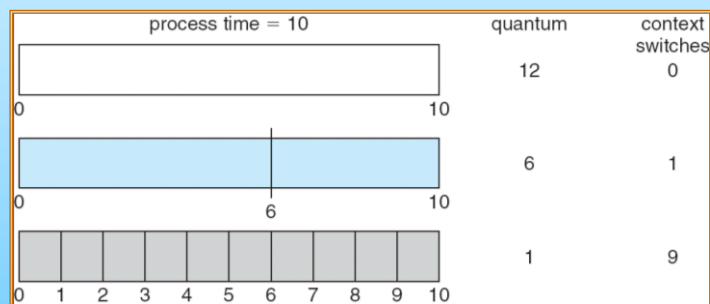
- Tempo medio di attesa:

$$(((77-20)+(121-97))+20+(37+(97-57)+(134-117)))+(57+(117-77)))/4 = (81 + 20 + 94 + 97) / 4 = 73 \text{ (mentre con SJF è 38)}$$

- Tipicamente si ha un tempo di completamento medio più elevato rispetto a SJF, ma si ha un miglior tempo di *risposta*

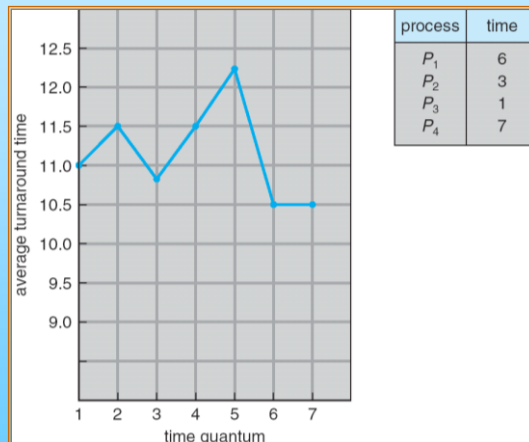
Sistemi Operativi A.A 2016/2017

Quanto di tempo e tempo per Context Switch



Sistemi Operativi A.A 2016/2017

Tempo di Completamento varia con il quanto di tempo



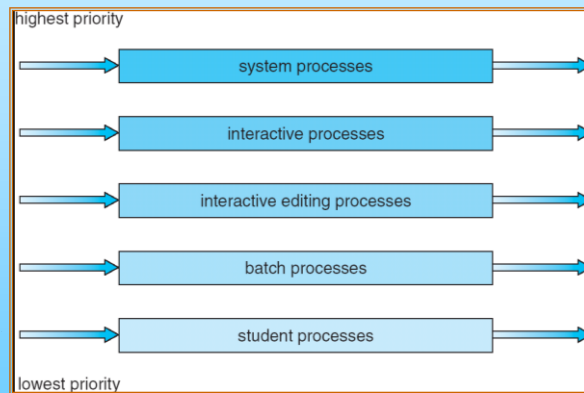
Sistemi Operativi A.A 2016/2017

Code multi-livello

- La coda dei processi pronti è divisa in code separate:
Esempio:
 - foreground (interattiva)
 - background (batch)
- Ogni coda ha il suo algoritmo di scheduling
 - foreground – RR
 - background – FCFS
- Scheduling deve essere fatto tra le code
 - Scheduling a priorità fissa; (es. serve tutti dalla coda di foreground quindi dal background). Possibile starvation.
 - Time slice – ogni coda prende un certo tempo di CPU che schedula tra i suoi processi; i.e., 80% a foreground in RR e 20% per background in FCFS

Sistemi Operativi A.A 2016/2017

Scheduling Code Multi-livello



Sistemi Operativi A.A 2016/2017

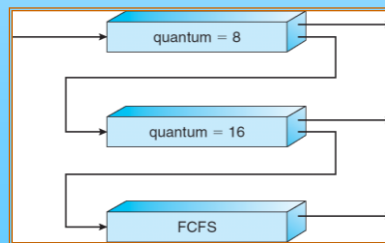
Code Multi-livello con retroazione

- Un processo si può spostare tra le varie code; l'invecchiamento può essere implementato così.
- Lo scheduler per code Multi-livello con retroazione definito tramite i seguenti parametri:
 - Numero di code
 - Algoritmo di scheduling per ogni coda
 - Metodo usato per determinare quando promuovere un processo
 - Metodo usato per retrocedere un processo
 - Metodo usato per determinare in quale coda un processo deve entrare quando il processo necessita di un servizio

Sistemi Operativi A.A 2016/2017

Esempio di Code Multi-livello con retroazione

- Tre code:
 - Q_0 – RR con quanto di tempo di 8 millisecondi
 - Q_1 – RR con quanto di tempo di 16 millisecondi
 - Q_2 – FCFS
- Scheduling
 - Un nuovo lavoro entra nella coda Q_0 . Quando ottiene la CPU il lavoro riceve 8 millisecondi. Se non finisce in 8 millisecondi, il lavoro e' spostato nella coda Q_1 .
 - In Q_1 il lavoro riceve 16 millisecondi. Se non viene completato, viene tolto dall'esecuzione e mandato nella coda Q_2 .



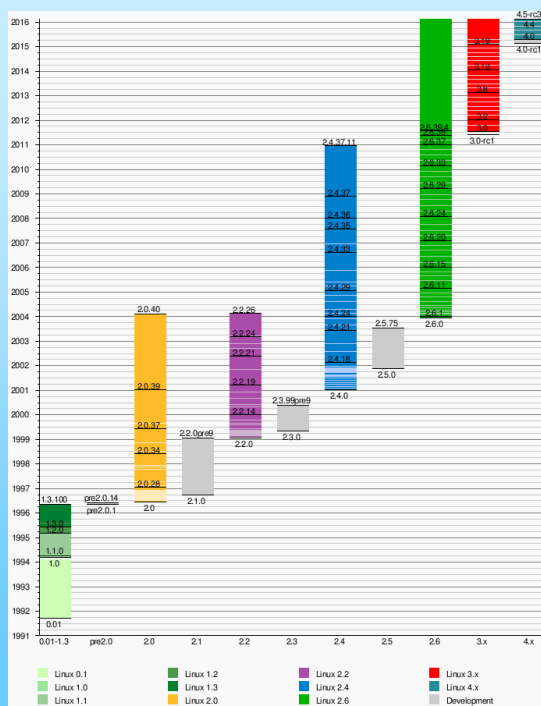
Sistemi Operativi A.A 2016/2017

Scheduling Linux

- Linux è un kernel OpenSource
 - sorgenti disponibili su <https://www.kernel.org/>
- Obiettivi:
 - timesharing
 - gestione dinamica delle priorità
 - tempi di risposta brevi
 - throughput elevato per processi in background
 - evitare starvation
 - poter gestire processi (soft) real-time (es. player multimediale)
- Prevede uno scheduling con prelazione su codice utente e dal kernel 2.6 anche il codice del kernel supporta la prelazione

Sistemi Operativi A.A 2016/2017

Linux timeline



Scheduling Linux

- Priorità statica e priorità dinamica
- priorità statica 1 – 99 usata per scheduling dei task (soft) real-time
 - con priorità statica i task possono essere schedulati:
 - SCHED_FIFO: quanto di tempo illimitato (lasciano la CPU solo se si bloccano, terminano, un task a più alta priorità diventa pronto)
 - SCHED_RR: soggetti a scheduling Round-Robin (nella stessa coda dei task FIFO)
 - conforme alla specifica POSIX
 - se un task FIFO entra in loop infinito sulla (unica) CPU il sistema può bloccarsi
- priorità statica 0, task schedulati SCHED_OTHER, usata per i processi normali, i task hanno una priorità dinamica, la politica di scheduling effettiva cambiata con diverse versioni del kernel

Scheduling Linux – Kernel 2.4

- Task a priorità 0
- Basato su 'epoche'
- Ad ogni epoca al processo viene assegnato un quanto di tempo (che consuma durante l'epoca)
- La priorità dinamica è legata alla durata del quanto
- Quando tutti i task in coda ready hanno terminato il loro quanto l'epoca è finita e vengono ricalcolati i quanti di tempo per l'epoca successiva, aumentando (della metà di quanto rimasto dall'epoca precedente) se processo non ha terminato il suo quanto (avvantaggiando i processi interattivi o I/O bound)
- Il task può variare il suo quanto/priorità impostando il **nice** value [-20 , 19] valori negativi aumentano la priorità (solo superuser può aumentare priorità)
- Per decidere quale task eseguire viene fatta scansione di tutti i task nella coda ready → $O(n)$

Sistemi Operativi A.A 2016/2017

Linux Scheduler O(1)

- Introdotto in kernel 2.6.8.1 (2004)
- Riduce il tempo di scelta del task da eseguire (su n tasks) portandolo da $O(n)$ a essere eseguito in tempo costante $O(1)$
- **140 livelli di priorità** (0-139, 0 priorità massima)
 - **0-99** per task real-time
 - **100-139** per task utente ($120 + \text{nice}$)
- Per ogni livello di priorità due liste FIFO
 - una con i task con quanto di tempo non esaurito (task attivi)
 - una con i task con quanto di tempo esaurito
 - Quando la lista dei task attivi è vuota le due liste vengono scambiate
- Trovare la priorità per cui esiste almeno un task attivo si fa in tempo costante rispetto al numero di task totale (implementato con un bit array)

Sistemi Operativi A.A 2016/2017

Linux Scheduler O(1)

- I task I/O bound vengono premiati e penalizzati quelli CPU bound con un incremento/decremento di priorità in range [-5,+5] (non per quelli real-time)
- basato su calcolo **sleep_avg** del task
 - incrementato del tempo in cui è stato in sleep (fino ad un massimo)
 - decrementato del tempo in cui ha usato CPU
- **bonus** = $(\text{sleep_avg} * \text{MAX_BONUS}) / \text{MAX_SLEEP_AVG} - \text{MAX_BONUS} / 2$
(MAX_BONUS = 10)
- il bonus non può far passare task non-RT a priorità dei task RT (0-99)
- Il timeslice dipende dalla priorità statica del task
 - maggiore priorità (valore più basso) timeslice più grande

Sistemi Operativi A.A 2016/2017

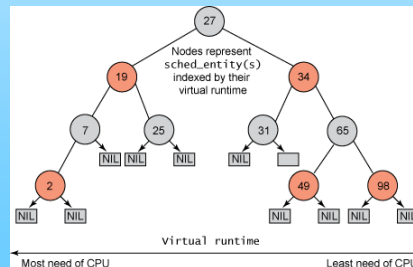
Linux Scheduler O(1)

- Interactivity credits
 - ottenuti quando task aspetta per tanto tempo
 - persi quando task usa tanta CPU
 - usati per non far perdere il bonus solo perché un task I/O bound usa occasionalmente tanta CPU
- complessità del codice gestione delle euristiche per bonus
- Nel Kernel 2.6.23 (2010) introdotto Completely Fair Scheduler CFS

Sistemi Operativi A.A 2016/2017

Linux Completely Fair Scheduler

- associa ad ogni task (non real-time) un **virtual_runtime** tempo di uso della CPU.
- per decidere quale task eseguire nella coda ready viene selezionato quello con **virtual_runtime minore** (quello che ha usato meno CPU)
- Questo viene fatto in tempo $O(\log n)$ usando alberi RB (red-black), alberi binari di ricerca bilanciati



Sistemi Operativi A.A 2016/2017

Linux CFS

- Il **virtual_runtime** aggiornato in base al nice level (-20 – 19) del task (priorità statica), tempo effettivo di CPU usata moltiplicato per (1.25^{nice}) , è come se si avesse:
 - CPU più veloce per task ad alta priorità (nice = -20)
 - CPU più lenta per task a bassa priorità (nice = 19)
- Un task viene selezionato ed eseguito, quando verrà fermato verrà aggiornato il suo virtual_runtime e se entra nella coda ready viene inserito nell'albero RB
- I task I/O bound tendono ad usare poca CPU e quindi tendono ad essere selezionati prima di quelli CPU bound

Sistemi Operativi A.A 2016/2017

Linux CFS

- Anche il quanto di tempo assegnato dipende dal nice level del task:
 - $\text{weight} = 1024 / (1,25 ^ \text{nice})$
 - $\text{nice} = 0 \rightarrow \text{weight} = 1024$
 - $\text{nice} = -1 \rightarrow \text{weight} = 1280 \dots \text{nice} = -19 \rightarrow \text{weight} = 71054$
 - $\text{nice} = 1 \rightarrow \text{weight} = 820 \dots \text{nice} = 20 \rightarrow \text{weight} = 12$
- Il quanto viene calcolato sulla base di tutti gli *weight* dei task nella coda ready
 - $\text{scheduling_period} = \max(n * \text{sched_min_granularity_ns}, \text{sched_latency_ns})$
 - $\text{time_slice}(T_k) = \text{scheduling_period} * \text{weight}(T_k) / (\text{SUM weight}(T_i))$
- All'aumentare della priorità (nice level basso) avrà un timeslice più grande rispetto agli altri task

Sistemi Operativi A.A 2016/2017

Classi di scheduling

- Con il kernel 2.6.23 è stato riorganizzato il codice per lo scheduling e definita interfaccia per algoritmi di scheduling.
- Permette di definire nel kernel più classi di scheduling ognuna gestita con il proprio algoritmo
- Le classi sono poste in sequenza di priorità:
 - STOP, per fermare il lavoro su CPU
 - RT, per processi real-time
 - FAIR, per processi normali
 - IDLE, per attività a bassissima priorità
- Più semplice aggiungere nuovi algoritmi di scheduling

Sistemi Operativi A.A 2016/2017