Sistemi Operativi

Laurea in Ingegneria Informatica
Università Roma Tre
Docente: Romolo Marotta

CPU scheduling

1. Tipi di scheduling
2. Politiche di scheduling
3. Multicore scheduling

Tipi di scheduling

Tramite l'analisi degli stati di un processo, si sono individuati tre classi di CPU scheduling:

- Long-term scheduling
  - Ammette nuovi processi al sistema
  - Controlla il livello di multiprogrammazione
  - Dipende dal carico del sistema
- Mid-term scheduling
  - Swap in/swap out di processi
  - Dipende dal livello di multiprogrammazione e memoria disponibile
- Short-term scheduling
  - Ammette processi ad andare in esecuzione sul processore
  - Invocato di frequente

Metriche per decisioni di scheduling

Lo scheduler può adottare diverse politiche di scheduling

Come comparare gli effetti delle politiche di scheduling?

- Uso di criteri per catturare specifici aspetti di interesse
    - Prestazionali
    - Non prestazionali
- Soggetti interessati
    - Sistema
    - Utente

Metriche per decisioni di scheduling

- Criteri **prestazionali** orientati **all'utente**
  - Tempo di risposta: tempo necessario affinché un processo inizi a produrre l'output
  - Tempo di turnaround: tempo totale tra la creazione di un processo ed il suo completamento
  - Deadline: percentuale di specifiche scadenze temporali rispettate
- Criteri **non prestazionali** orientati **all'utente**
  - Prevedibilità: variazione ridotta del tempo di risposta o di turnaround

Metriche per decisioni di scheduling

- Criteri **prestazionali** orientati **al sistema**
  - Throughput: numero di processi completati per unità di tempo
  - Utilizzazione: percentuale di tempo in cui una determinata risorsa risulta impegnata
- Criteri **non prestazionali** orientati **al sistema**
  - Fairness: evitare starvation di processi
  - Priorità: favorire processi con maggior priorità
  - Bilanciamento delle risorse: evitare il sottoutilizzo di risorse e favorire processi che non richiedono risorse in sovraccarico

Preemptive e non-preemtive scheduling

Non-preemptive scheduling:

- Una azione di scheduling **attende** che il processo abbandoni la cpu
  - Processo terminato
  - Bloccato per I/O

Preemptive scheduling:

- Una azione di scheduling può attendere che il processo abbandoni la CPU o **interrompere** il processo in esecuzione indipendentemente dalle attività che esso sta svolgendo
  - All’occorrenza di un interrupt
  - All’occorrenza di un evento (e.g., terminazione di I/O)
  - Invocazione di syscall

First Come First Serve (FCFS)

- Non-preemptive
- I processi in stato ready vengono eseguiti nell'ordini in cui sono stati inseriti nella ready-to-run queue

Pros:

- Semplice
- No starvation

Cons:

- Non garantisce minimo turnaround time medio o waiting time medio
- Soggetto ad una forte variabilità
- Non massimizza l'utilizzo delle risorse

Processo	Tempo di CPU
P1	2
P2	6
P3	1

AVG turnaround = (1 + 3 + 9)/3 = 4.33

AVG waiting time = (0 + 1 + 3)/3 = 1.33

AVG turnaround = (6 + 8 + 9)/3 = 7.66

AVG waiting time = (0 + 6 + 8)/3 = 4.66

First Come First Serve (FCFS)

CPU
I/O

CPU-bound:
- P1

I/O bound:
- P2,P3,P4,P5

Arrival:
P1,P2,P3,P4,P5

Convoy effect: tanti processi "leggeri" aspettano un unico processo cpu-intensive

P1
P2
P3
P4
P5

CPU idle

I/O idle

CPU idle

Shortest Job First (SJF)

- Non-preemptive
- I processi vengono schedulati in accordo a quanto tempo occuperanno la CPU
- Anche chiamato Shortest Process First (SPF) o Shortest Next CPU-burst First

Processo	Tempo di CPU
P1	2
P2	6
P3	1

Pros:

- Minimizza turnaround/waiting time
- Favorisce il throughput

AVG turnaround = (1 + 3 + 9)/3 = 4.33

AVG waiting time = (0 + 1 + 3)/3 = 1.33

Cons:

- Possibilità di starvation
- Necessità di predire per quanto tempo un processo riesede in CPU

Avg: $S_{n+1} = \frac{1}{n}\sum_{i=0}^{n} T_i$

Exp Avg: $S_{n+1} = \alpha T_n + (1 - \alpha) S_n$

α vicino ad 1 => maggior peso ad osservazioni recenti => maggior instabilità

Può supportare prelazione (Shortest Remaining Time Next - SRTN):

- L'arrivo di un processo ready con minor lunghezza stimata di CPU-burst

Round Robin (RR)

- Preemptive
- Ai processi viene assegnato un quanto di tempo (time quantum o time slice)

Pros:

- No starvation
- Attesa massima limitata: $(n-1)q$

Cons:

- La taglia della time slice è critica per le performance
- Unfair: sfavorisce processi I/O bound

Processo	Tempo di CPU
P1	2
P2	6
P3	1

AVG turnaround = (6 + 8 + 9)/3 = 7.66

AVG waiting time = (0 + 6 + 8)/3 = 4.66

Q = 2

AVG turnaround = (9 + 4 + 8)/3 = 7

AVG waiting time = (3 + 2 + 2)/3 = 2.33

Round Robin (RR)

La taglia della time slice è critica per le performance

- Q maggiore del massimo CPU burst ⇒ RR collassa su FIFO
- Q minore o uguale al tempo di context-switch
  - ⇒ Il processore spende più del 50% del suo tempo ad eseguire context-switch
- Q minore del tempo necessario a compiere un'unità di lavoro (e.g. attivazione di I/O)
  - ⇒ Maggiori tempi di attesa e sottoutilizzo dispositivi di I/O

Processo	Tempo di CPU
P1	2
P2	6
P3	1

Q = 6

AVG turnaround = (6 + 8 + 9)/3 = 7.66

AVG waiting time = (0 + 6 + 8)/3 = 4.66

Q = 2

AVG turnaround = (9 + 4 + 8)/3 = 7

AVG waiting time = (3 + 2 + 4)/3 = 3

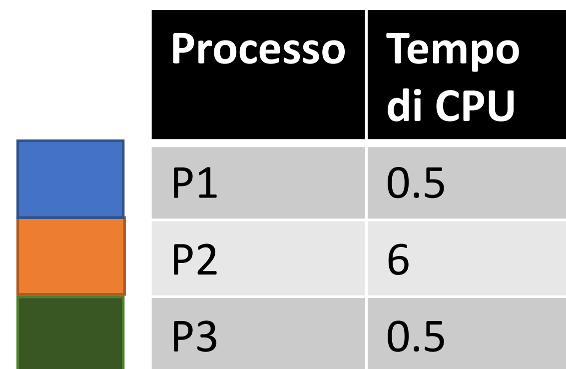
Round Robin (RR)

Unfair: sfavorisce processi I/O bound

- Processi I/O bound non utilizzano tutto il loro quanto
- Processi CPU bound tendono a rientrare immediatamente nella ready queue

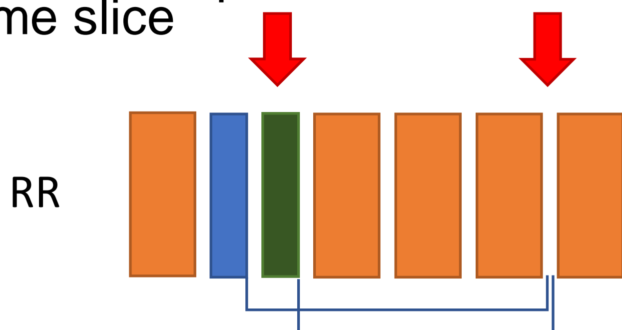
Soluzione: Virtual Round Robin (VRR)

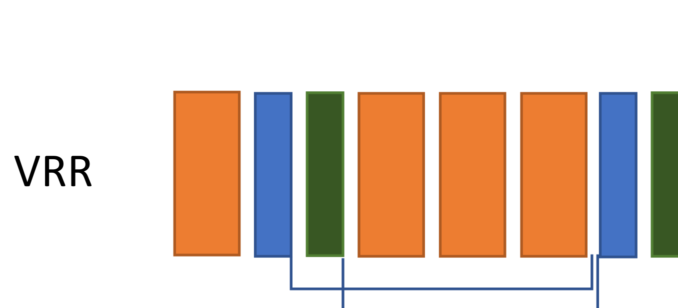
- I processi hanno un credito per non aver speso tutto il quanto
- I processi ready con credito hanno una coda ausiliaria dedicata
- Lo scheduler sceglie il processo prima dalla coda ausiliaria e poi dalla ready-to-run queue
- Un processo proveniente dalla coda ausiliaria esegue al più per un tempo pari al suo credito, ossia il tempo non utilizzato della precedente time slice



Processo	Tempo di CPU
P1	0.5
P2	6
P3	0.5

- Q = 1
- P2 CPU-bound, P1 e P3 I/O bound





Priority Scheduling

- Una priorità viene assegnata ai processi
- Il dominio delle priorità è un insieme su cui esiste un ordinamento totale
- I processi vengono schedulati secondo priorità decrescenti (prima i processi a priorità più alta)
- Non è necessario che priorità più alte abbiano valori maggiori rispetto a priorità più basse
  - Esempio
    - Dominio: numeri interi positivi incluso lo 0
    - Priorità 0 = priorità massima
- Tipicamente processi a medesima priorità vengono serviti con politica FCFS o RR

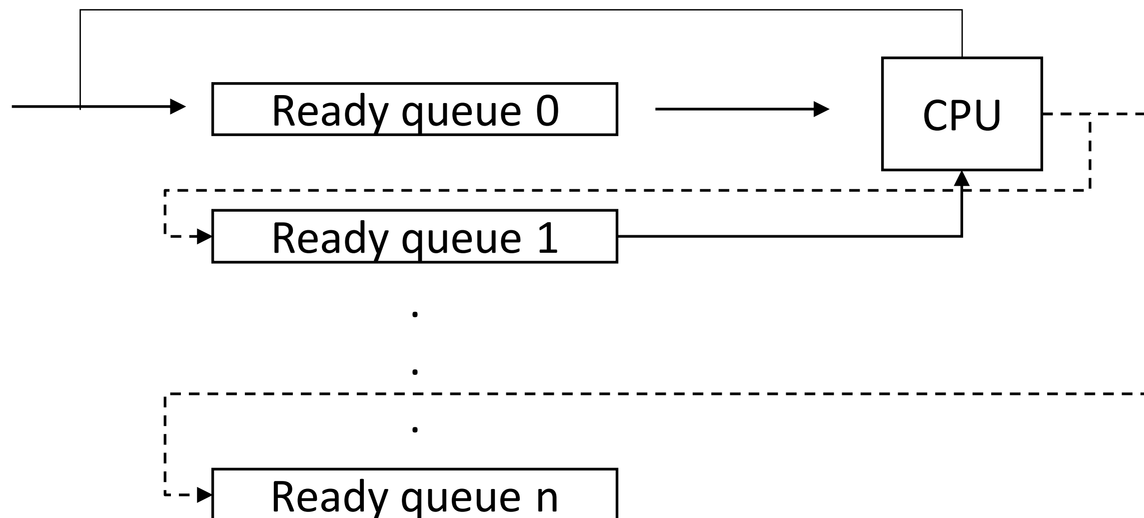
Priority Scheduling

FCFS, SJF, SRTN sono specifici schemi a priorità:

- FCFS
  - Dominio: tempo
  - Assegnazione: istante di tempo di inserimento in coda
  - Selezione: processo con valore di priorità minore
- SJF
  - Dominio: tempo
  - Assegnazione: durata di CPU-burst
  - Selezione: processo con valore di priorità minore
- SRTN
  - Dominio: tempo
  - Assegnazione: durata di CPU-burst residua
  - Selezione: processo con valore di priorità minore

Multilevel Feedback Queue

- Più code FCFS o RR
- Ad ogni coda è associata una priorità
- Si schedulano processi in una coda a priorità più bassa quando tutte le code a priorità più alta sono vuote
- Se un processo utilizza tutto il suo quanto verrà spostato in una coda con minor priorità
- Processi a priorità bassa possono soffrire di starvation
  - Problema parzialmente alleviato se la time slice è crescente per priorità decrescenti



Priority Scheduling

Processi a priorità bassa possono soffrire di starvation

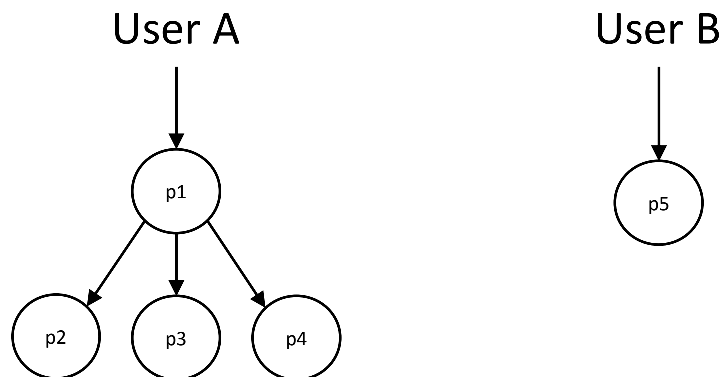
- Dipende da come sono definite ed assegnate le priorità
  - Vedi SJF, SRTN, MFQ
- Meccanismi di **aging**, ossia di invecchiamento, per far salire le priorità di processi a bassa priorità (CPU-bound)
  - Highest Response Ratio Next (HRRN)
    - Dominio: Reali positivi
    - Assegnazione: $p = \frac{w+s}{s}$ dove w è il tempo speso in coda e s è la previsione di durata del prossimo CPU-burst
    - Selezione: max(p)
  - MFQ
    - Introduzione di politiche per migrare processi da code a bassa priorità verso coda ad alta priorità

Ancora sulla fairness

- Le soluzioni discusse prendono decisioni basate sul singolo processo come fossero unità indipendenti l'una dall'altra
- Nonostante l'uso appropriato di feedback sulle priorità (no starvation) è ancora possibile una forma di unfairness

RECALL:

- I processi possono creare altri processi



Ipotesi:

- p1,p2,p3,p4,p5 CPU-BOUND
- Scheduling RR

- User A: 80% CPU
- User B: 20% CPU

Soluzione:

- Assegnare ad un gruppo di processi uno *share* di CPU
- Lo share è poi assegnato ai processi appartenenti al gruppo
- Il meccanismo può essere esteso con l'utilizzo delle priorità

- User A: 50% CPU
- User B: 50% CPU

- p5: 50% CPU
- p1,p2,p3,p4: 12.5% CPU

Fair-Share scheduling

- I processi vengono suddivisi in n gruppi
- Ciascun gruppo k ha un peso $W_k \in (0,1)$ e $\sum_{k=1}^{n} W_k = 1$
- $BASE_i$ è una priorità preassegnata alla creazione del processo i
- CPU_i è l'utilizzo di CPU da parte del processo i nell'ultimo intervallo di tempo
- $GCPU_k$ è l'utilizzo complessivo di CPU da parte dei processi appartenenti al gruppo k nell'ultimo intervallo di tempo
- Al processo i del gruppo k viene associata una priorità pari a:

$$P_i = BASE_i + \frac{CPU_i}{2} + \frac{GCPU_k}{4 \cdot W_k}$$

- Valore di priorità minore ≈ priorità elevata

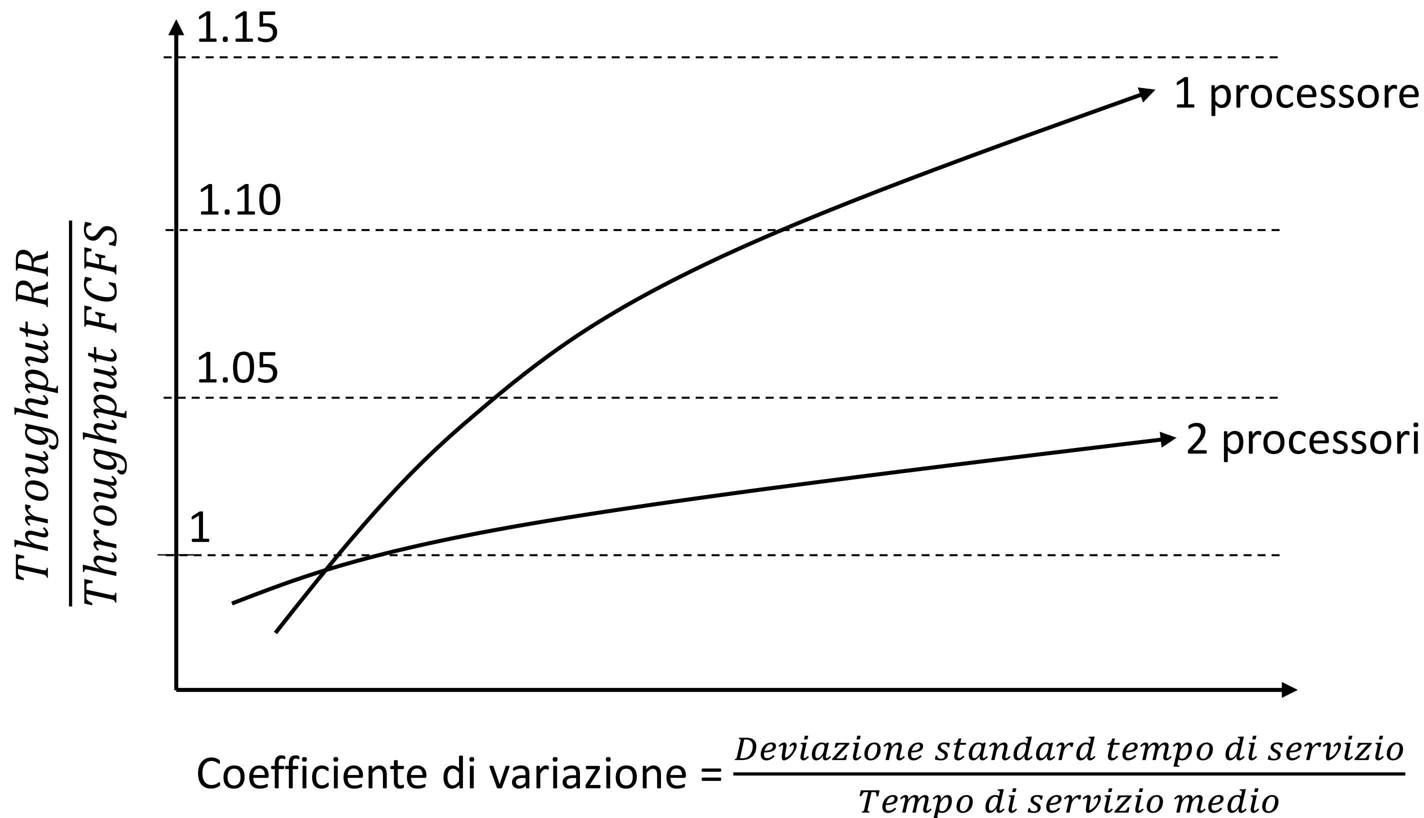
Un processo CPU-bound fa scendere di priorità i processi del proprio gruppo di un ammontare proporzionale al peso del gruppo

Multiprocessor scheduling

- Molteplici CPU (o CPU-core) condividono la memoria principale
- In un sistema strettamente accoppiato tutte le unità di calcolo sono controllate da un unico sistema operativo
- Nuove problematiche:
  - Con quali politiche?
  - Come assegnare processi/thread ai processori?

Multiprocessor scheduling

- Con quali politiche?
  - Utilizzo di policy classiche (FCFS, RR, …)

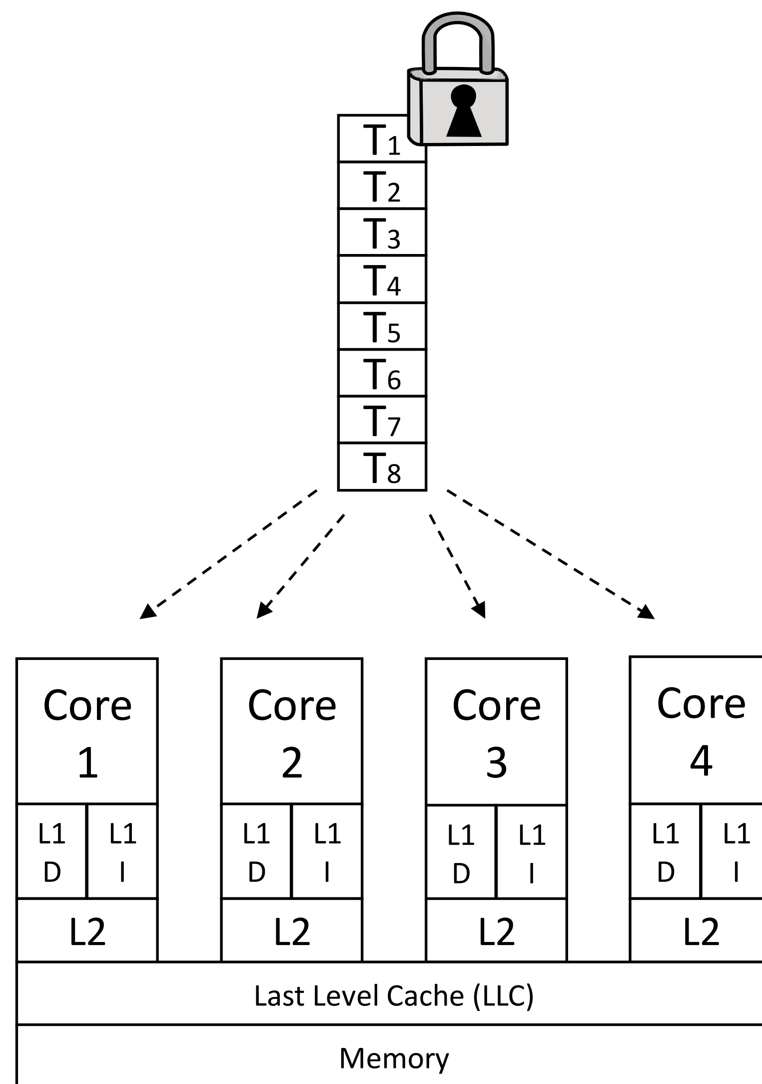


Multiprocessor scheduling

- Come assegnare processi/thread ai processori?
- Assegnazione statica:
  - Per tutta la durata del thread/processo, questo va in esecuzione sul medesimo processore
  - Pros: overhead ridotto
  - Cons: possibilità di sottoutilizzo dei processori
- Assegnazione dinamica:
  - un thread/processo può esser eseguito su diversi processori durante la sua esecuzione
  - Pros: migliora l'utilizzo dei processori
  - Cons: overhead maggiore dovuto alla migrazione da un processore all'altro

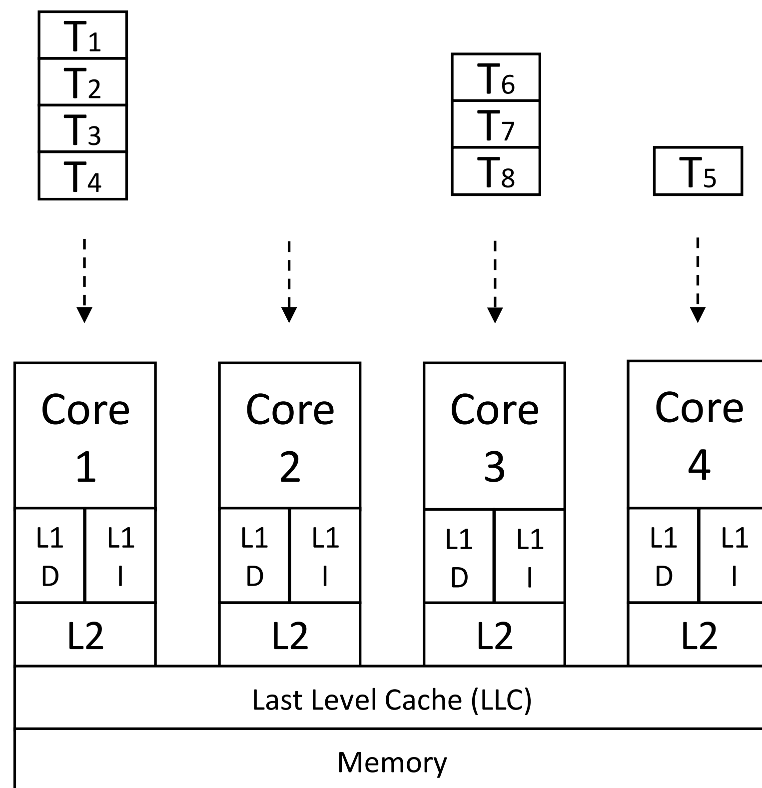
Load sharing

- Il carico (ossia thread/processi da eseguire) è condiviso fra tutti i processori
  - Ready queue condivisa
- Pros:
  - Il carico è distribuito uniformemente
- Cons:
  - Necessità di sincronizzare gli accessi alle strutture dati condivise
  - Ridotto effetto delle cache



Load balancing

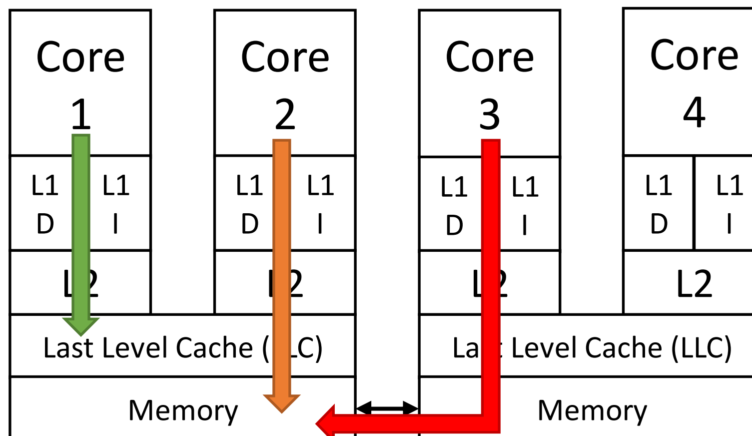
- Il carico (ossia thread/processi da eseguire) è preassegnato ai processori
  - Ready queue per processore
- Pros:
  - Non serve sincronizzazione tra i processori per manipolare le code
  - Miglior utilizzo delle cache
- Cons:
  - Necessità di bilanciare periodicamente il carico per ciascun processore



Affinity

La latenza di un’istruzione che accede a memoria è variabile:

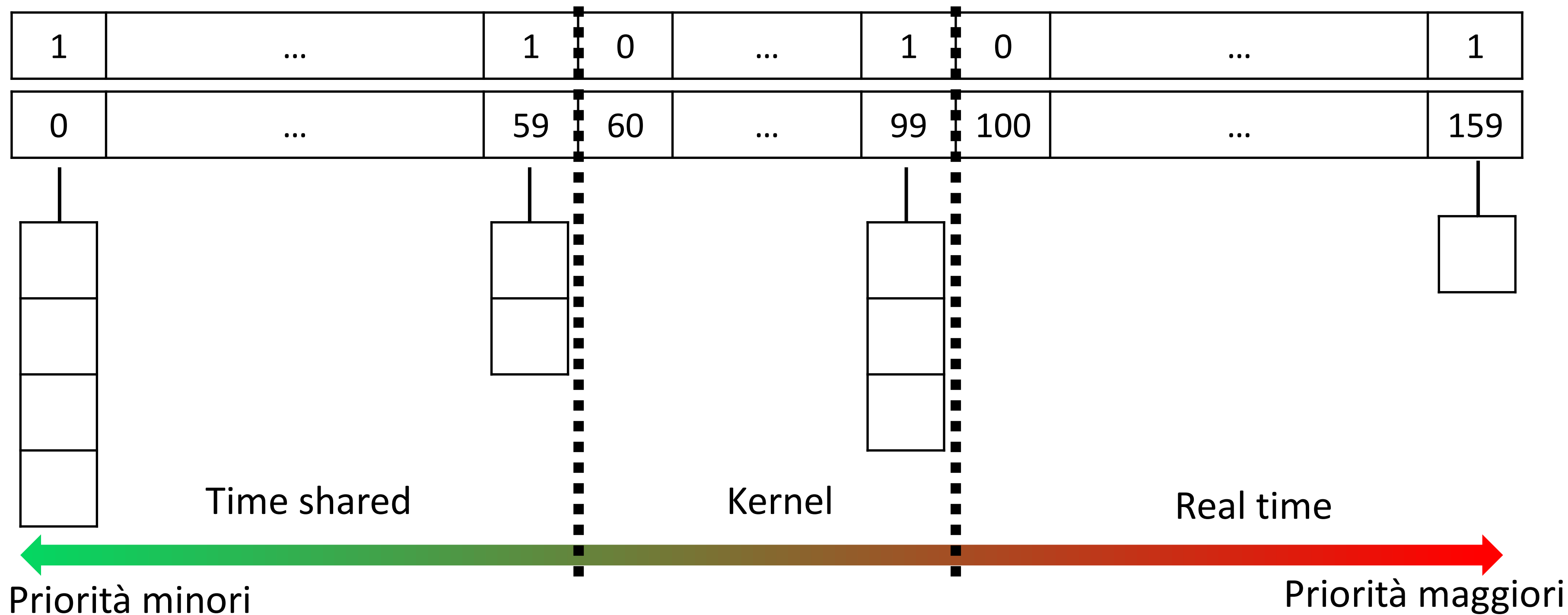
- cache hit vs cache miss
- accesso ad un banco di memoria locale vs remoto



Non-Uniform Memory Access (NUMA)

UNIX System V Release 4

- 160 livelli di priorità
- Ad ogni priorità corrisponde una coda gestita in RR
- Bitmap per individuare rapidamente le code non vuote
- Tre classi di priorità
  - Real time: time slice e priorità fissata
  - Time shared: time slice e priorità variabile
  - Kernel: processi in modo kernel, interrompibili su preemption point



Linux

Due tipi di scheduling:

- Real time:
  - Priorità statiche (da 0 a 99)
  - Suddiviso in ulteriori due classi
    - FIFO (SCHED_FIFO) – interrotti se:
      - Un task con più priorità diventa ready
      - Il task è in attesa di un evento
      - Il task rilascia volontariamente la cpu con apposita syscall
    - Round Robin (SCHED_RR):
      - Come FIFO, ma con time-slicing
- Non-real-time (SCHED_OTHER o SCHED_NORMAL):
  - Priorità dinamiche

Linux O(n)-scheduler

Kernel < 2.6

- Load sharing (coda globale)
- Kernel non-preemptive (supporto debole per task RT)
- Time slice media significativa (task interattivi penalizzati)
- MFQ
  - Variazione dello scheduling UNIX tradizionale
  - Ciascuna coda in RR
  - $P_i = BASE_i + \frac{CPU_i}{2} + nice_i$
- Costo O(n) per operazione di scheduling (n = # task nel sistema)

Linux O(1)-scheduler

2.6 <= Kernel < 2.6.23

- Load balancing (run queue per processore)
- Kernel preemptive (miglior supporto per task Real Time)
- Time slice media ridotta rispetto al precedente (processi interattivi penalizzati)
- MFQ
  - Due array: active e expired
  - Ciascuno associato ad una bitmap (come UNIX SV4)
- Costo O(1) per operazione di scheduling (indipendente rispetto al #task nel sistema)

Linux Completely Fair Scheduler

Kernel >= 2.6.23

- Associa ad ogni task un **virtual run time:**
  - pari al run time per task con nice = 0
  - minore del run time per task con nice < 0
  - maggiore del run time per task con nice > 0
- Task ready mantenuti in un albero bilanciato (red-black tree)
  - O(logn) per individuare il task con priorità maggiore
  - Introdotta una cache software per individuare rapidamente il task a massima priorità
- La time slice assegnata ad un task non ha taglie predefinite, ma dipende:
  - dal nice assegnato a ciascun task
  - dalla **targeted latency** impostata nel sistema
  - dalla taglia minima assegnabile
- Supporto al grouping
  - Gruppi di task vengono considerati come un unico task nelle assegnazioni delle priorità e time slice (vedi Fair share scheduling)

Linux Completely Fair Scheduler

Kernel >= 2.6.23

- Load balancing:
  - Ripartizione gerarchica delle risorse in domini
  - Bilanciamento all’interno di un dominio
  - Più il carico su domini allo stesso livello è sbilanciato e più è probabile che un task venga migrato di dominio