02-05-2023

Obiettivi degli algoritmi di scheduling

Per progettare un algoritmo di schedulig è cruciale **cosa** deve fare il SO e **quali sono i suoi obiettivi** e questi ultimi dipendono dall'ambiente (*macchina applicativa*). Si distinguono vari ***ambienti di lavoro***:

1. **BATCH**: sono sistemi in cui la velocità della risposta è **meno importante**. Risulta fondamentale **l'esecuzione LUNGA del processo**. Di conseguenza sono **più adatti gli algoritmi non preemptive** e quindi vengono applicati maggiormente negli ambienti *INDUSTRIALI*. (*potrebbe esserci anche un algoritmo preemptive MA con una **LUNGA ESECUZIONE***)
   - In questo caso si **riducono** il numero di *cambi di contesto*
2. **INTERATTIVI**: sono macchine che interagiscono con l'utente. Di conseguenza **la velocità della risposta** alle richieste ricevute ha un'**ALTA PRIORITA'**. Classici esempi sono l'apertura di un sito web, di una cartella o simili. In questo caso risultano più adatti gli **algoritmi preemptive**.
3. **REAL-TIME**: E' determinante l'esecuzione di un processo in un **preciso istante di tempo** (classico esempio è la *catena di montaggio*) e **NON è sempre utile** usare metodi **preemptive**. In questo caso non si usa un algoritmo di scheduling come gli altri perchè ogni singolo processo sa di per sé che verrà eseguito per periodi non troppo lunghi

A prescindere dall'ambiente di lavoro usato, gli algoritmi di scheduling hanno degli obiettivi comuni:

1. **equità nell'assegnazione della CPU**: processi comparabili, cioè *appartenenti alla stessa categoria* e quindi che hanno una **stessa priorità**, devono avere un uguale trattamento. Di conseguenza, per sempio, in un *sistema interattivo*, tutti i processi che *interagiscono con l'utente*, devono essere comunque eseguiti **prima** di qualsiasi altro processo
2. **bilanciamento** nell'uso delle risorse hardware: **tutte le parti** del sistema devono essere **impegnate** e bisogna **evitare** che la CPU venga occupata da *processi inattivi*.

> Domanda ->*Come si fa a valutare se si è sviluppato un algoritmo di scheduling o uno scheduler buono o non buono?*

Metriche delle prestazioni

Se l'algoritmo è applicato in un sistema **BATCH**, si devono verificare tutte le seguenti metriche allo stesso tempo:

- **MASSIMIZZARE** il numero di processi che vengono **eseguiti e completati** in **un'unità di tempo** (***THROUGHPUT*** O *produttività*)
- **MINIMIZZARE il tempo di *turnaround***, cioè il **tempo medio** di completamento ed è il tempo fra **l'istante di arrivo della richiesta** del processo (e quindi il suo *accodamento*) e l'**istante del suo completamento**
- **MINIMIZZARE** il **tempo di attesa** ciopè il **tempo medio** che il processo mediamente trascorre fra i processi **READY**

> *Fra le 3 metriche si ha un **peso maggiore** sul **tempo di attesa** e su questa metrica vi è una **maggiore influenza** da parte dell'algoritmo di scheduling*

> Inoltre, *massimizzare il throughput* **non vuol dire** necessariamente *minimizzare il tempo di turnaround*

Se l'algoritmo è applicato in un sistema **INTERATTIVO**, si devono verificare tutte le seguenti metriche allo stesso tempo:

- E' fondamentale rispondere nel minor tempo possibile alle richieste dell'utente
- Bisogna quindi **MINIMIZZARE** il **tempo di risposta**, cioè il tempo che passa fra *richiesta ricevuta* e l'*esecuzione effettiva della richiesta*

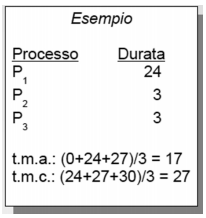
Se l'algoritmo è applicato in un sistema **REAL-TIME**, si devono verificare tutte le seguenti metriche allo stesso tempo:

- Si devono **RISPETTARE LE SCADENZE** dell'avvio o della terminazione di un processo in modo da permettere la **regolarità di esecuzione**
- **PREVEDIBILITA'** dei tempi di esecuzione dei processi

Algoritmi di Scheduling in sistemi BATCH

First-Come First-Served (*FCFS*)

- Si prende una coda (coda dei processi *READY*) e si mandano in esecuzione i processi **IN BASE ALL'ORDINE DI ARRIVO**. Questo sistema si chiama **First-Come First-Served** (*FCFS*). Questo tipo di algoritmo risulta penalizzante per i processi I/O bounded.
- E' un algoritmo **NON-PREEMPTIVE** e quindi i processi rilasciano autonomamente la CPU
- Se, durante l'esecuzione, arrivano altre richieste, queste ultime verranno aggiunte in coda e poi eseguite fino allo svuotamento della coda



`t.m.c.` = tempo di completamento medio = 27

- I processi che arrivano in coda si aspettano a vicenda -> Per completare P3 si deve aspettare P1 e P2.
  `t.m.a.` = tempo medio di attesa = 17
- Il primo processo attende 0 secondi per essere eseguito -> P2 aspetta il tempo di completamento di P1 (24).

Questo algoritmo è semplice da implementare ma **penalizza fortemente i processi I/O**. In questo caso il **disco risulta essere inattivo** perchè **non viene usato**

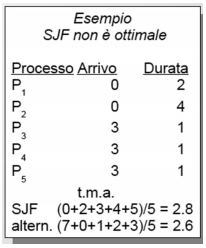
Shortest Job First (*SJF*)

Un altro algoritmo è **Shortest Job First** (*SJF*):

- Si eseguono prima i processi che richiedono **MENO DI ESECUZIONE** e in questo modo si hanno **tma più brevi**
- Bisogna conoscere, all'inizio, i tempi di esecuzione dei processi

- E' un algoritmo **NON-PREEMPTIVE** ed è ottimale **ESCLUSIVAMENTE** se i **lavori** sono tutti **subito disponibili** ma comunque resta facile da implementare

> Se si riprende il primo esempio si ha un $tmc = 13$ e un $tma = 3$



Esempio
SJF non è ottimale

Processo	Arrivo	Durata
P_1	0	2
P_2	0	4
P_3	3	1
P_4	3	1
P_5	3	1

	t.m.a.
SJF	(0+2+3+4+5)/5 = 2.8
altern.	(7+0+1+2+3)/5 = 2.6

In questo caso:

- tempo di completamento t.c = :
  - p1 = 2
  - p2 = 6
  - p3 = 6+1 **-3** = 4 (**perchè p3 non è arrivato al tempo 0**)
  - p4 = 7+1 **-3** = 5
  - p5 = 8+1 **-3** = 6
- tempo di attesa t.a = 2.8:
  - p1 = 0
  - p2 = 2
  - p3 = 2+4 **-3** = 3
  - p4 = 6+1 **-3** = 4
  - p5 = 7+1 **-3** = 5

Dalla versione alternata, cioè partendo da p2 e poi p1,p3,p4,p5 allora si ha:

- il tempo di attesa $t.a$ = **2.6**:
  - p2 = 0 (*dura 4 e nel frattempo arrivano p3,p4,p5*)
  - p3= 1
  - p4=2
  - p5=3
  - p1=7

Quindi, quando *i processi arrivano **durante l'esecuzione***, allora questo algoritmo **NON** è più ottimale perchè, invertendo il due processi, allora si è ottenuto un risultato migliore

Esempio: Dati 4 processi dove ognuno impiega tempo a, b, c, d, il tmc di $P1 = a$, $P2 = a + b$, $P3 = a + b + c$, $P4 = a + b + c + d$. In generale, con ordine `P1 P2 P3 P4` viene $t.m.c = \frac{4a+3b+2c+d}{4}$

Shortest Remaining Time Next (*SRTN*)

Un altro algoritmo è **Shortest Remaining Time Next** (*SRTN*)

- si basa sulla politica SJF (dare priorità ai processi che durano di meno)
- è un algoritmo **PREEMPTIVE**
- Inizialmente, quando si hanno tutti i processi a tempo 0, si **applica la politica SJF**
- Se nel frattempo arrivano altri processi, il SO guarda il tempo dei nuovi processi e, se il tempo è più basso, allora si esegue quello che ha tempo più corto. Il tempo tolto va riconfrontato in futuro

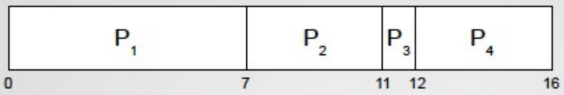
> *Nell'esempio di prima si esegue P1 e poi P2. Quando parte P2, dipo 1 ms arrivano P3 P4 e P5. P2 ha ancora 3 ms di lavoro ma i processi nuovi ne hanno 1 ciascuno. P2 viene stoppato e vengono eseguiti i nuovi processi. P2 verrà poi confrontato con il tempo rimanente (3 ms)*

Confronto fra i 3 algoritmi con l'uso del Diagramma Gantt (*ESAME*)

Si considerano 4 processi che arrivano in tempi diversi:

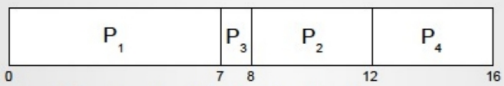
Processo	Arrivo	Durata
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

FCFS:



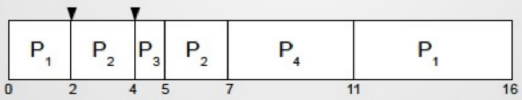
- tempi di attesa: P_1=0; P_2=5; P_3=7; P_4=7 (media 4.75);
- tempi di completamento: P_1=7; P_2=9; P_3=8; P_4=11 (media 8.75);

SJF:



- tempi di attesa: P_1=0; P_2=6; P_3=3; P_4=7 (media 4);
- tempi di completamento: P_1=7; P_2=10; P_3=4; P_4=11 (media 8);

SRTN:

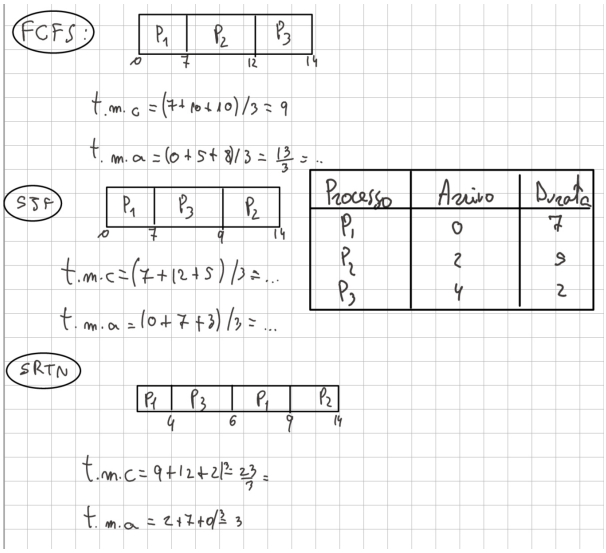


- tempi di attesa: P_1=9; P_2=1; P_3=0; P_4=2 (media 3);
- tempi di completamento: P_1=16; P_2=5; P_3=1; P_4=6 (media 7).

In particolare, in *SRTN*, quando si calcolano i **tempi di attesa**, si devono sottrarre anche i tempi che i processi hanno **consumato**. Invece, nel calcolo del tempo di completamento, si devono **sottrarre i millisecondi in cui il processo in questione non era presente.**

L'algoritmo **SRTN risulta essere il più efficiente** rispetto agli altri 2

ESERCIZIO:



04-05-2023

Scheduling nei sistemi interattivi

Round-Robin (*RR*)

- E' un algoritmo di scheduling **PREEMPTIVE** *FCFS* (*First Come First Served*)
- Assegna un quanto di tempo a un processo.
  - Appena scade il tempo assegnato, il processo **preleva** tale processo e lo rimuove.
- Se **il processo si blocca** prima (*I/O*) verrà anche rimosso
- Si basa su una **coda di processi** e il primo che si trova in testa è il successivo da eseguire (*concetto di* `Queue`)
  - Se il processo **B termina il suo quanto di tempo**, viene prelevato e rimesso in fondo alla coda

> *Quanto tempo si deve assegnare (**timeslice**) a ogni processo?*
> *Se il timeslice è troppo breve, allora, allora aumento il numero di cambi di processi e di contesto (switch) e si spreca tempo in cambi*
> *Se il timeslice è troppo lungo, allora riduco il numero di cambi di processi (switch) e di contesto. In questo caso incremento il tempo di attesa fra processi.*

In media uno switch dura 1ms mentre il timeslice è 20-50ms

Bisogna valutare 2 concetti che influiranno:

- `context switch` = scambio di modalità fra utente e kernel
- `process switch` = scambio fra processi

> *Se i switch sono molti, si ha un grande spreco di tempo perchè, chiaramente, ogni switch richiede del tempo.*

- Più si **aumenta il timeslice** e più **riduco gli switch fra processi** e quindi un processo resta **in attesa per molto più tempo**.
- Quindi risulta deterministico **ricavare il giusto timeslice** per aumentare le prestazioni dell'algoritmo

> *Quando il timeslice finisce, allora interviene un **INTERRUPT DI CLOCK*** sul processo.

Se il timeslice è "corretto" allora ci sono dei vantaggi e svantaggi:

- +E' **semplice da implementare** (*basta usare una **coda di processi***) e ad ogni processo si assegna il *timeslice*
- -**Tutti i processi sono uguali** fra loro a **livello di importanza** e questo non è possibile in caso di sistema interattivo
  - *Un processo per email non è uguale a un processo che richiede la digitazione di testo su schermo*

Algoritmo di scheduling con priorità

Bisogna eseguire uno **scheduling a priorità** (i processi demoni o simili (*per esempio*) devono avere priorità minore rispetto a una richiesta di I/O). Quindi serve **differenziare i processi**:

- I processi con **alta priorità** devono avere una **maggior possibilità di essere scelti** se si trovano nella coda dei processi `READY`
    - Con questo sistema, però, si ha una **difficoltà** nella gestione dei processi con **priorità bassa**. Se ci sono sempre processi con priorità alta, **quelli con priorità bassa** rimangono **in attesa all'infinito**

L'idea è: quando scade il timeslice, si va a scegliere il prossimo processo con priorità alta. **Ad ogni clock** si deve **RIDURRE LA PRIORITA'** del processo in esecuzione e in questo modo si garantirà che anche i processi di bassa priorità verranno eseguiti.
![[]]

Priorità statiche e dinamiche

Le priorità possono essere **STATICHE** (`nice`) o **DINAMICHE** ($\frac{1}{f}$) e per assegnarle/modificarle si usa `nice` (Unix) su un processo e si usa solo da `root` perchè è un comando delicato.

> *Per vedere i dettagli del comando si usa* `man nice`

Con l'uso della *priorità dinamica* si usa dare **priorità maggiore ai processi I/O bounded**.

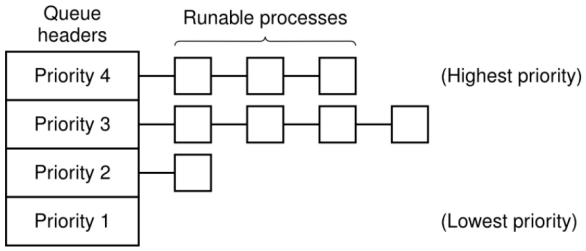
Si usa anche assegnare una priorità in base al timeslice consumato f in particolare $\frac{1}{f}$. Si può fare anche $\frac{timeslice\_complessivo}{timeslice\_consumato}$.

> *Generalmente meno timeslice se ne consuma, più alta e la priorità e viceversa. **In questo modo** si assegnano priorità **DINAMICHE***

SJF si può rivedere con un algoritmo di priorità dove **più lunga è l'esecuzione, più bassa è la sua priorità** dove, in generale, si considera **l'inverso della durata del processo**. In questo senso SJF può essere visto come un **ALGORITMO DI PRIORITA'.**

Classi di priorità

Se i processi hanno la **stessa priorità**, allora c'è un sistema che suddivide i processi in **CLASSI DI PRIORITA'**: in una singola classe ci sono i processi in una coda (*FIFO*) che hanno la stessa priorità.



Se si esegue un processo di priorità 3 e ne arriva uno di priorità 4 allora si deve prelevare (*PREEMPTIVE*) ed eseguire il nuovo processo

Problema: i processi di classi inferiori rischiano di non essere mai eseguiti (***STARVATION***) e una possibile soluzione è detta ***AGING*** che consiste nell'***aumentare gradualmente la priorità dei processi che attendono*** a lungo in coda nella stessa classe.

I processi possono avere **task differenti** (*processo interattivo vs processo background*) e quindi *si distingue* ulteriormente *per gruppi* dove vengono individuate le categorie di ogni processo in questo modo **si distingue l'attività specifica del processo**

All'interno delle singole classi (quindi ***a partità di priorità***), si ha una selezione dei processi che devono essere eseguiti utilizzando lo scheduling **ROUND-ROBIN** (*eseguo per prima i processi che si trovano in testa alla coda*).

> *Si può usare un sistema di assegnazione del timeslice differente per ogni categoria di processo*

09-05-2023

Per gli algoritmi di scheduling con priorità di distinguono algoritmi preemptive e non preemptive.

Per differenziare i processi interattivi dai processi in background si potrebbe agire nel seguente modo:
Scheduling con coda multipla: la coda dei processi pronti si suddivide in:
- coda dei **processi interattivi** -> algoritmo roundrobin
- coda dei **processi in background** -> algoritmo FCFS
- Si possono applicare scheduling differenti sulle due code (con priorità su quelli interattivi)

Si può anche giocare sul timeslice:

- 80% del tempo dedicato ai processi interattivi
- 20% del tempo dedicato ai processi in background

Gestione dei processi interattivi ed eventuali algoritmi

SRTN permette di superare il limite di SJF ma se quest'ultimo conosce a priori tutti i tempi, risulta essere un algoritmo ottimale.

- Si usa SJF nella gestione dei processi interattivi: si basa su un approccio `attesa della richiesta -> esecuzione della richiesta`
- I comandi più brevi vengono eseguiti all'inizio ma comunque si deve sapere quale, fra tutti i processi, è il processo più breve.

Il problema è l'identificazione della durata del successivo processo. Si potrebbe fare con una **stima**. E si stima sulla base delle esecuzioni precedenti svolte (***burst precedenti***).

Shortest Process Next

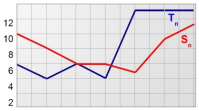
- Tenta di simulare SJF in processi interattivi
- Il processo successivo da eseguire viene stimato tramite la media esponenziale delle lunghezze (durate) delle precedenti esecuzioni di occupazione della CPU.

Si definiscono:

- T_n -> lunghezza dell'n-esimo CPU-burst
- S_{n+1} tempo richiesto per il prossimo CPU-burst
- a -> valore fisso fra 0 e 1 e **pesa**, nella **stima**, le **un ricordo di quello che è successo** nel *breve/medio/lungo* tempo.

Lastima del processo n+1 è:

$$S_{n+1} = S_n(1-a) + T_n a$$



- se $a = 0$ vuol dire che la storia recente non ha alcun effetto e quindi ottiene $S_{n+1} = S_n$

- se $a = 1$ vuol dire che il recente CPU-burst è quello cruciale che determina la stima del prossimo ma non viene considerata la cronologia di quello che è successo in precedenza e quindi si ottiene $S_{n+1} = T_n$
- Si da un peso ponderato ad $a = \frac{1}{2}$ fra la cronologia passata e l'evento presente.
- Il problema è il punto di partenza S_0. Si può rendere costante

> *In generale: le scelte per il bilanciamento del carico che si prendono sono basate sull'esperieza passata e presente dei fatti*

Funzionamento SPN

- Ci si trova nel caso n dove $S_n = 10$ mentre $T_n = 6$
- Ne stimo 8 ma in reatà ne ha fatti solo 4

T_n	6	4	6	4	13	13	13
S_n	10	8	6	6	5	9	11

- Usualmente **a = 1/2** e questo algoritmo stima il tempo del **CPU-burst del successivo processo** e in questo caso si considera una **cronologia di media lunghezza**

> *Se il tempo effettivo di un processo T_n è più alta della stima, quest'ultima si incrementa (ovviamente)*

Algoritmo di Scheduling Garantito

E' un approccio diverso e **stabilisce una percentuale di uso della CPU** che deve essere **RISPETTATA**.

- Va **calcolata la percentuale di uso di CPU spettante per ogni processo**: considera quanto tempo è stato utilizzato dal timeslice è stato usato e, in base a questa quantità, determina quanta quantità di CPU un processo ha diritto ad utilizzare. Viene scelto il processo con rapporto minore
- Inoltre, il rapporto deve essere minore rispetto al consumo
- fare promesse reali e mantenerle utenti connessi avranno 1/n della potenza CPU (idem processi)
- tenere traccia quanta CPU ricevuta
- (Tc / n) = tempo dalla creazione diviso n
- l'algoritmo esegue il processo con rapporto minore

Algoritmo di Scheduling a Lotteria

- **Ogni processo ha un biglietto assegnato** e quando si deve decidere chi deve occupare la CPU, l'algoritmo **estrae** un **numero casuale** e il processo che va in esecuzione è il processo che ha il numero precedentemente assegnato
- Una volta che il **processo estratto va in esecuzione**, viene **consumato** il numero assegnatoli
- In caso di **esigenze diverse** si possono **assegnare biglietti extra** ad un processo e vuol dire **aumentare la probabilità** che il processo venga scelto dopo l'estrazione.
- I processi possono **cooperare**: se `A` si trova nella sezione critica, `B` non può accedere. `A` prende i suoi e li cede a `B` in modo da aumentare la probabilità che `B` venga estratto e viceversa verrà fatto da `B`

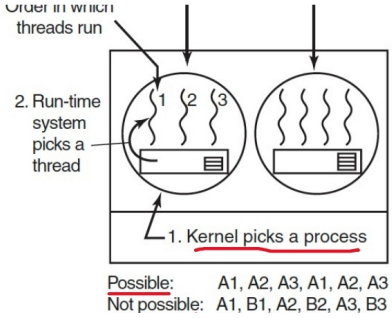
Algoritmo di Scheduling Fair-Share

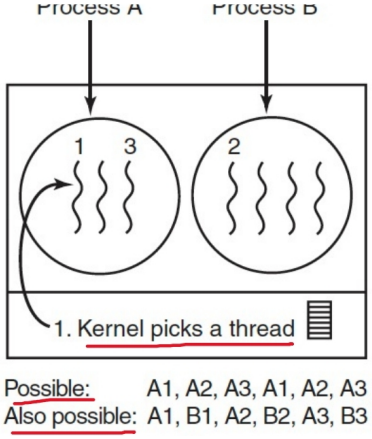
- *Se l'utente A esegue 9 processi mentre l'utente l'utente B ne esegue 1, allora la CPU è dedicata quasi interamente ad A e questa non è una corretta distribuzione della CPU.*
- Questo algoritmo **tiene in considerazione i proprietari dei processi** e determina **un'equo uso della CPU fra gli utenti del sistema** a prescindere dal numero di processi che ogni singolo utente esegue
- Si assegna una **proporzione di uso della CPU** ai vari utenti **in base al numero di processi da eseguire**

> *Classicamente questi algoritmi non vengono usato ma si usa, come detto prima, l'algoritmo ROUND-ROBIN in caso di parità di priorità (stessa classe di priorità)*

Scheduling dei Thread

I thread possno essere implementati in vari e i pro dell'uno corrispondono ai contro dell'altro e viceversa:

- **A livello Utente** (*il **kernel conosce i thread** e se uno di essi si blocca, allora viene bloccato l'intero processo. Il cambio fra thread è veloce*):
  - 

  - Lo **scheduling** è **PERSONALIZZATO** e sceglie di **assegnare** un ***timeslice*** da assegnare **al processo stesso**
  - Il **processo stesso** usa un **sistema di scheduling** con il quale fa l'assegnazione dei tempi ai singoli thread (*distribuendo il timeslice assegnato al processo*)
  - *Interrupt di clock* fra thread **non sono possibili** quindi i thread lavorano in modalità **non preemptive** e ognuno di loro *lasciano la CPU quando finoscono interamente il tempo assegnato* (*e questo rappresenta un aspetto negativo e un limite*).
  - Di conseguenza *potrebbe capitare* che un thread **consuma l'intero timeslice** assegnato al processo
- **A livello Kernel** (*il kernel riconosce i thread quindi può stoppare il singolo thread se necessario. Un thread di un processo potrebbe essere cambiato con un thread di un altro processo*):
  - 

  - Il **timeslice è assegnato al thread** dal kernel
  - Adesso lo **scheduling dei thread è gestito dal kernel**

- I **thread vengono considerati tutti uguali** e ciò potrebbe **compromettere le performance** e quindi si potrebbe perdere tempo nel **cambio di contesto**
- Piuttosto che **sostituire un thread con un altro di un altro processo**, si sfrutta l'informazione del processo e si cerca di **forzare**, se possibile, lo **scambio** fra thread **con un altro thread dello stesso processo** (*per evitare di scambiare thread fra processi diversi*)

Differenze fra Thread Utente e Thread Kernel (scheduling)

Le prestazioni sono dfiferenti:

- Il **cambio di contesto** è più lento
- A livello kernel il thread bloccato non sospende il processo -> non c'è livello utente
- Il grosso vantaggio dei Thread Utente è il permesso di usare uno ***SCHEDULING PERSONALIZZATO*** (*quindi dipendente dal tipo di applicazione*)

Di norma si usa un sistema **MISTO**.

Gestione scheduling in un sistema multiprocessore

Esistono diversi modi per gestire lo scheduling di un multiprocessore:

1. ***ASIMMETRICO***: si hanno diversi processori di cui **UNO** si occupa dello scheduling, elaborazione I/O, smistamento dei processo -> ***MASTER SERVER***. Gli altri, ***SLAVE*** eseguono i processi.
2. ***SIMMETRICO***: Ogni processore esegue un determinato processo e ogni processore fa la stessa identica cosa. Esiste una coda e un algoritmo di scheduling che smista i processi fra i processori. La ***coda dei processi pronti*** è **UNICA** (il kernel gestisce lo scheduling. si deve evitare che 2 processori si aggiudicano la CPU) per tutti i processori oppure **DIVERSA** per ogni processore (si cerca di *sfruttare al massimo la CACHE dello **stesso processore** per migliorare i **tempi di esecuzione***

Politiche di scheduling

- Si cerca di eseguire i processi negli stessi processori per permettere di usare i dati della cache di quel processore e quindi evitando di trasportare dati dalla memoria principale.
- Presenza o assenza di predilezione per i processori. Un processore cerca di prediligere un particolare processo e può essere:
  - ***DEBOLE***: il sistema **tenta di eseguire un processo** in quel determinato processore dove è stato eseguito in precedenza ma **non lo garantisce**
  - ***FORTE***: Il sistema **forza e garantisce** che un processo vada in esecuzione in un **determinato processore** (*o insieme di processori*)

> *In generale il sistema SIMMETRICO viene usato su Windows/Linux.*

Bilanciamento del carico

Lo scheduling deve eseguire il **bilancio fra processi** e processori che hanno al proprio interno delle code (visto che non c'è un'*unica coda*).

- Il carico di lavoro deve essere ugualmente distribuito
- si applica una pseudo-migrazione:
  - un processore che ha una coda piena, allora un processo migra verso un altra coda più vuota di un altro processore.

Si distinguono:

- ***migrazione guidata*** (push): un'attività (*demone*) che periodicamente controlla il carico di lavoro di ciascun processore. Quando vi è uno squilibrio di attività e carichi di lavoro, allora si ridistrbuisce in maniera equilibrata i carichi di lavoro ed effettua la migrazione su un processore la quale coda di processi è più vuota.
- ***migrazione spontanea*** (pull): la coda di un processore diventa vuota. Spontaneamente questa coda viene riempita

> *Spesso questi 2 approcci sono usati insieme in parallelo* (*UNIX*)

Il bilanciamento del carico di lavoro e la predilezione del processore sono due politiche contrastanti e quindi bisogna trovare un compromesso in base alle attività. Non esiste una regola dove si predilige una di queste politiche ma in generale si cerca di eseguire un mix fra le politiche (*ove possibile*)

Cosa usano i nostri Sistemi Operativi

Elementi comuni: thread, SMP, gestione priorità, predilezione per i processi I/O bounded
Windows:

- scheduler basato su code di priorità con varie euristiche per migliorare il servizio dei processi interattivi e in particolare di foregrount
- evita il problema dell'inversione delle priorità
  Linux:
- Scheduling basato su task (generalizzazione di processi e thread) usato con **alberi rosso-neri** ordinati in base al tempo di esecuzione:
  - Si garantisce un corretto bilanciamento grazie alla **proprietà dei *RB-Tree***
- Moderno scheduler garantito e si basa su *Completely Fair Scheduler* (CFS)
- Non considera processo/thread o altro ma sono tutte ***TASK***
  MacOS:
- scheduler basato su code di priorità (*Mach Scheduler*)