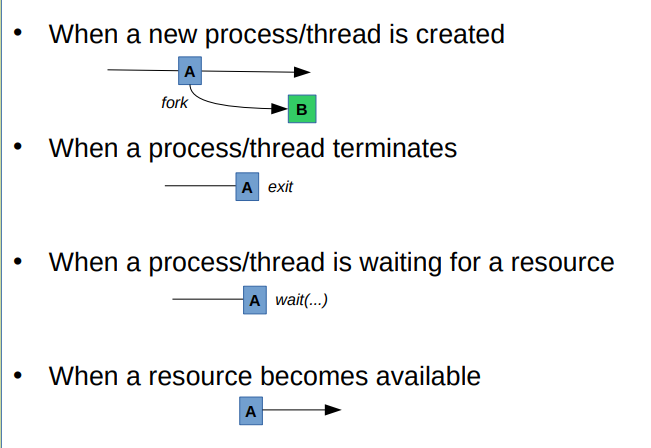
**Schedling**

In un sistema multi programmato lo scheduler determina l’ordine di ottenimento delle risorse del sistema da parte dei processi e thread.

Lo scheduling può essere

* Cooperative multitaskjng: ogni processo coopera con gli altri rilasciando la cpu dopo x tempo
* Preemptive multitasking: il sistema gestisce il rilascio della cpu per ogni processo

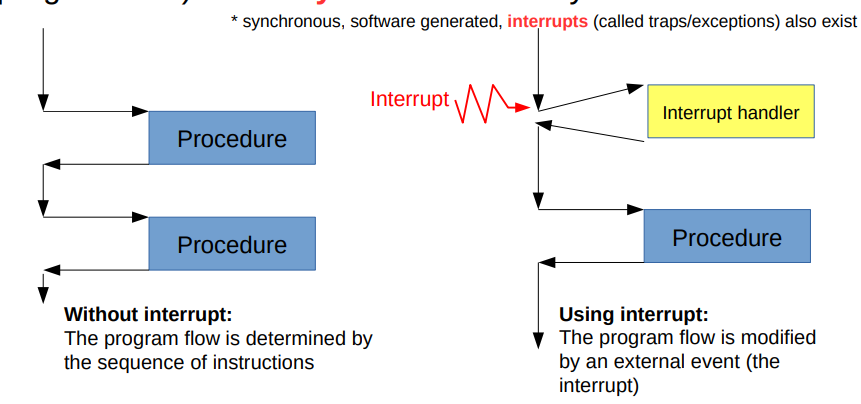
Altri casi in cui viene chiamato lo scheduler



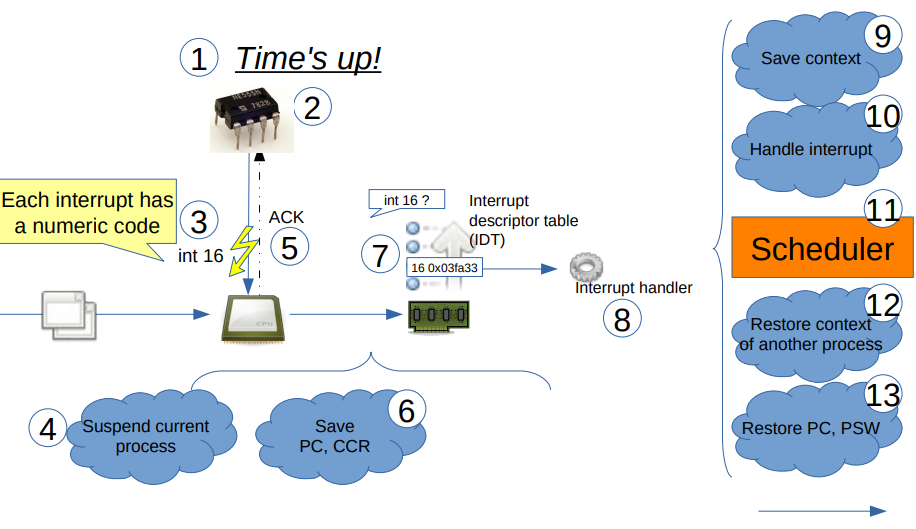
Il comportamento dei processi influenza lo scheduler:

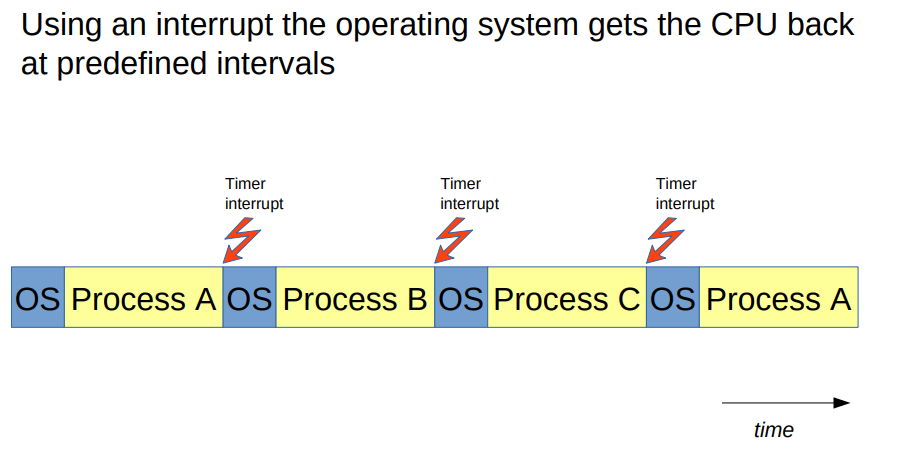
* Processi I/O Bound : sono processi con frequenti input e output, quindi numerose interruzioni (i programmi GUI sono I/O Bound)
* Processi CPU Bound : sono processi dove l’uso della cpu è intenso, quindi lo scheduler per equità deve bloccare il processo dopo un x tempo.

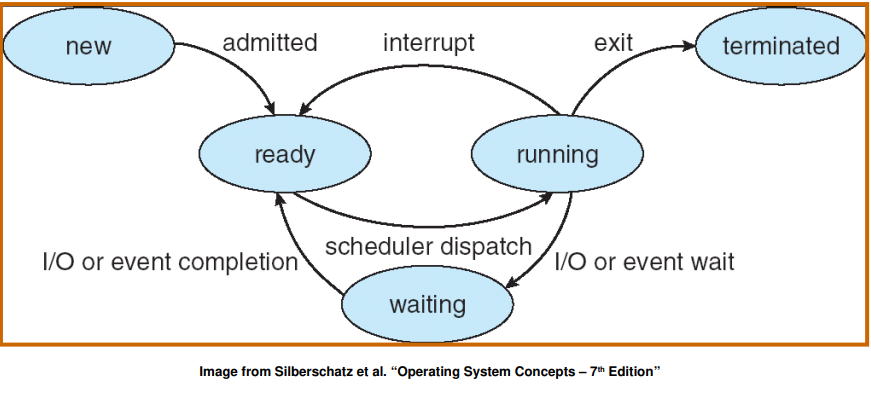
Gli interrupt ha un tempo stabilito, viene gestito lato hardware, quando scade il tempo blocca la cpu e effettua lo scheduling.

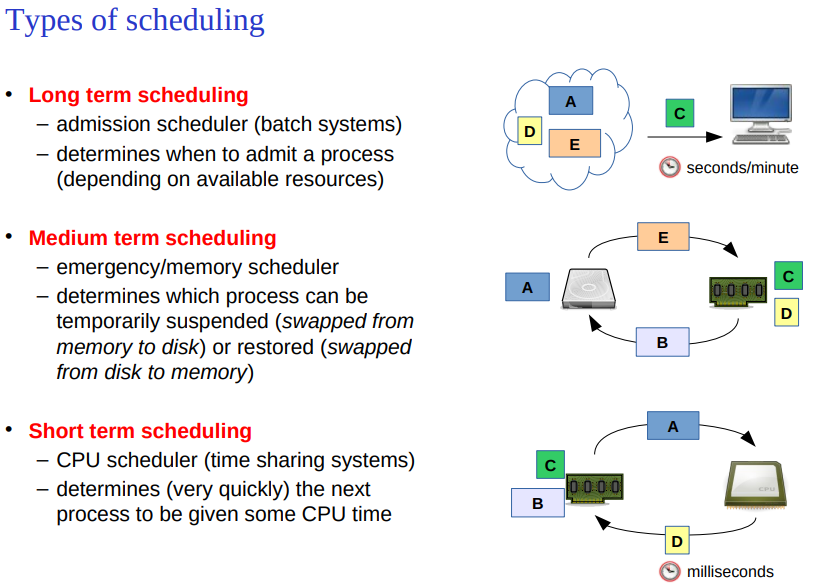


Ad ogni interrupt è associata una routine specifica implementata dal sistema operativo, chiamata interrupt handler.









Nel medium term scheduling usa la memoria swap (memoria su disco) per liberare memoria per altri processi che devono essere eseguiti.

**Ambienti di computazione**

* Batch:
  + Lunghi tempi di computazione
  + Nessuna interazione utente
* Interattivo:
  + Più utenti allo stesso tempo
* Real time (lo scheduler deve garantire un certo tempo di risposta)

**Come valutare uno scheduler**

* Throughput (Sistemi batch)
  + Numero di lavori per unità di tempo
* Turnaround time (sistemi batch)
  + Tempo medio richiesto per completare un lavoro
* Response Time (sistemi interattivi)
  + Tempo tra la ricezione di un comando e l’ottenimento del risultato
* Predicibilità e regolarità (sistemi realtime)
  + Il tempo di risposta deve essere garantito

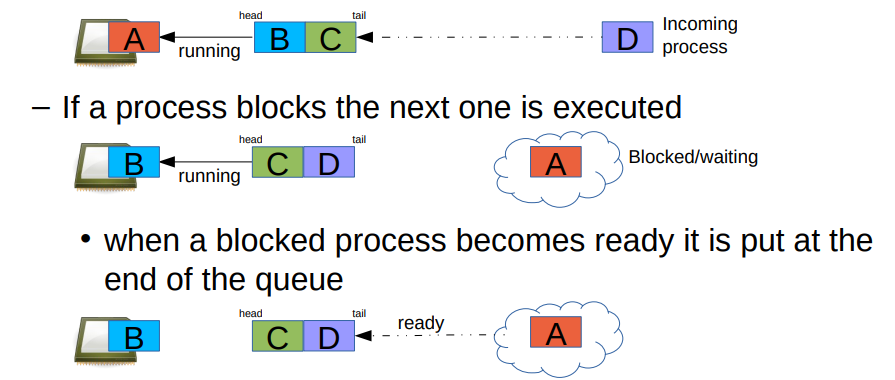
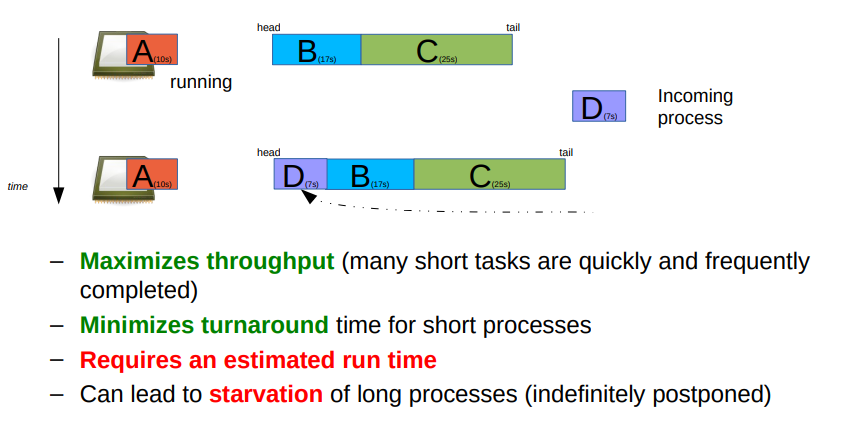
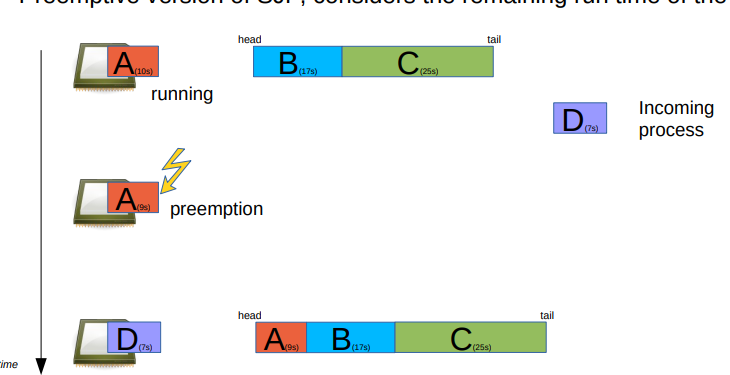
Solitamente Diversi ambienti --> diversi obbiettivi

Ma ci sono degli obbiettivi comuni negli scheduler:

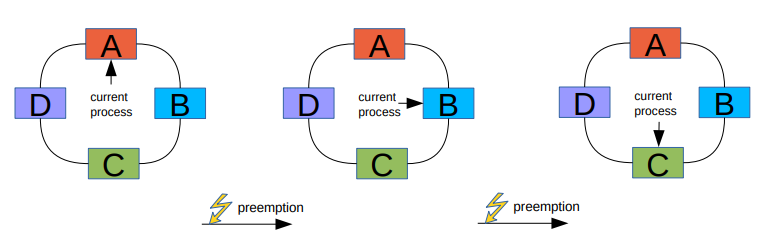
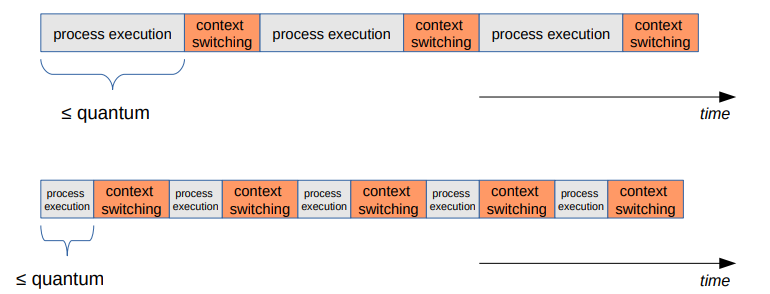
* Equità(Fairness)
  + Tutti i processi devono ottenere in modo equo l’uso della CPU
* Policy enforcement
  + Le decisioni di scheduling devono essere rispettate
* Bilanciamento
  + Massimizzare l’uso delle risorse

**Algoritmi di scheduling**

**Batch scheduling:**

* **FCFS (First come First-Served) o FIFO (first in first out**
  + Ogni processo per quanto necessita
  + I nuovi lavori vengono messi in coda
  + 
* **SJF (Shortest Job First)**
  + Vengono priorizzati i lavori brevi (stima)
  + 
* **SRTN (Shortest Remining Time Next)**
  + Da priorità ai processi che hanno tempo tempo di esecuzione rimanente minore(variante di SJF)
  + 

**Sistemi interattivi**

* **Round Robin**
  + I processi sono organizzati in una lista circolare
  + Lo scheduler da per ogni processo uno slot di tempo (quantum)
    - Quanto il quntum scade si da la cpu a un altro processo in stato ready
  + 
  + Se il quantum è grande Il tempo di risposta è molto alto
  + Ridurre il quantum si può, ma deve determinare anche il tempo dei context switch
  + 
* Si può introdurre una priorità per i processi in base al processo, non è semplice implentarlo, andrebbe cambiata in modo dinamico

**Come implementare una priorità**

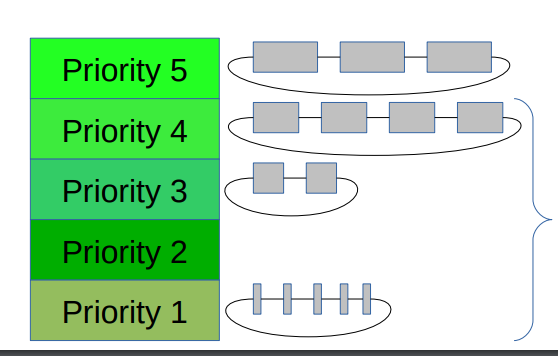
**Code multiple**

Si crearno delle code per ogni priorità, si eseguono prima le priorità più alta a scendere liberando le liste per ogni livello.

Si rischia starvation se si hanno tanti processi di priorità alta

**Liste circolari per ogni livello di priorità**

Processi con alta priorità ottengono un quantum più alto, i processi con minor priorità avranno un alto tempo di risposta (pessimo per scheduler interattivo)

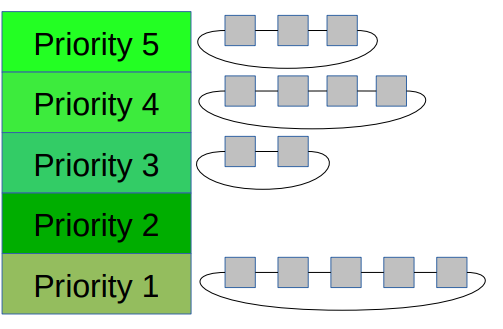


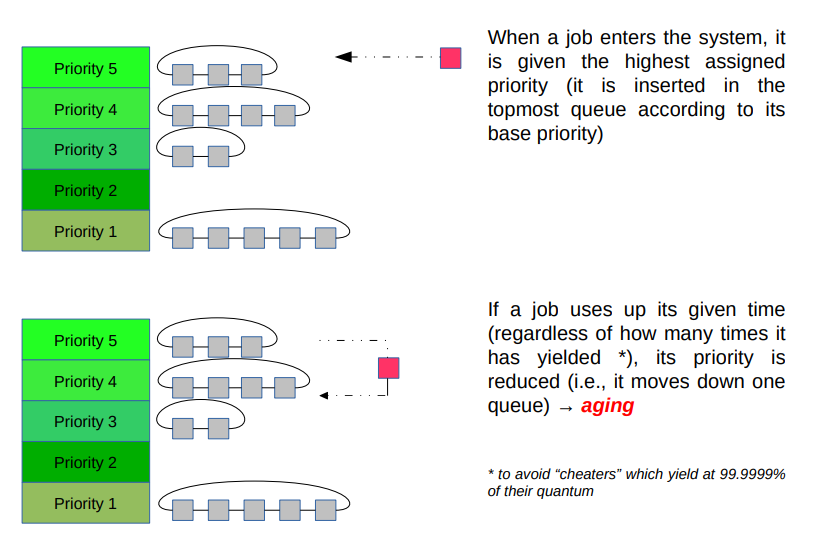
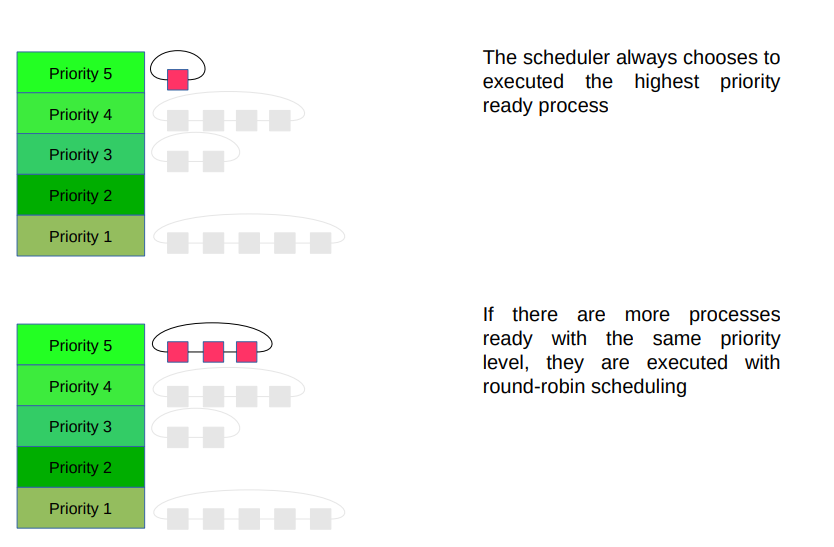
**MLFQ (Multilevel Feedback Queue)**

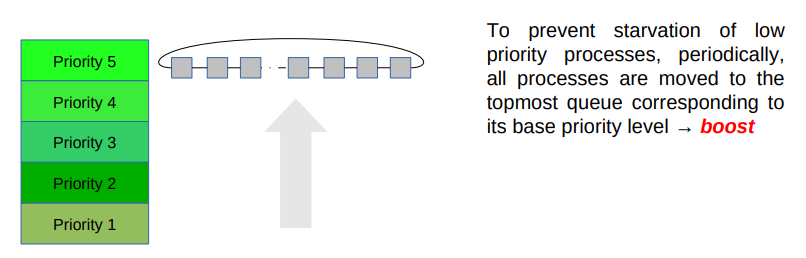
Ogni livello di priorità è linkato in una coda round-robin

Finché ci sono processi pronti nella coda con priorità più alta, eseguili, altrimenti passa a una coda di classe con priorità più bassa

Dipende da comportamento, muove i processi nelle varie code.







Periodicamente i processi vengono riportati alla priorità originale.

Il feedback è dato dal tempo che impiega a cambiare livello

**Lottery Scheduling**

Assegna ad ogni processo un N>1 numero di tickets, proporzionale alla priorità del processo stesso.

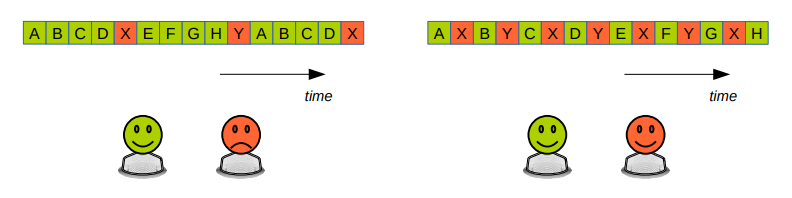
Periodicamente effettua l’estrazione e il processo vincente viene eseguito.

I vantaggi:

* Priorità dinamica
* I processi possono dare tickets ad altri processi (esempio client-server)

**Fair-Share Scheduling**

È una policy che divide il tempo di CPU tra gli utenti, assegnare una priorità agli utenti indipendentemente dal numero di processi



È possibile combinare algoritmi Fair-Share con tutti gli altri.

Gli altri algoritmi usano policy che guardano ai processi e non agli utenti.

**Sistemi multicore**

I sistemi multicore hanno una memoria condivisa che fornisce risorse ai vari thread e processi

Questi tipi di sistemi introducono il problema della località

Località:

* **Temporale**: quando un dato è accessibile, sarà probabilmente riutilizzato in un futuro vicino
* **Spaziale**: un dato all’indirizzo di memoria x probabilmente verranno usati anche i dati vicini ad esso

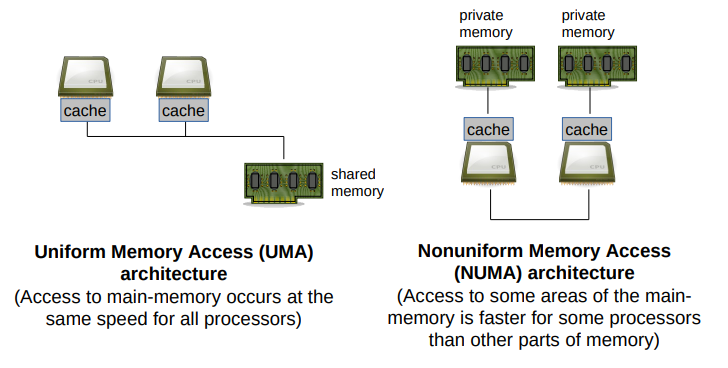
Quando un thread è eseguito per istante di tempo la cache corrispondente conterrà pi

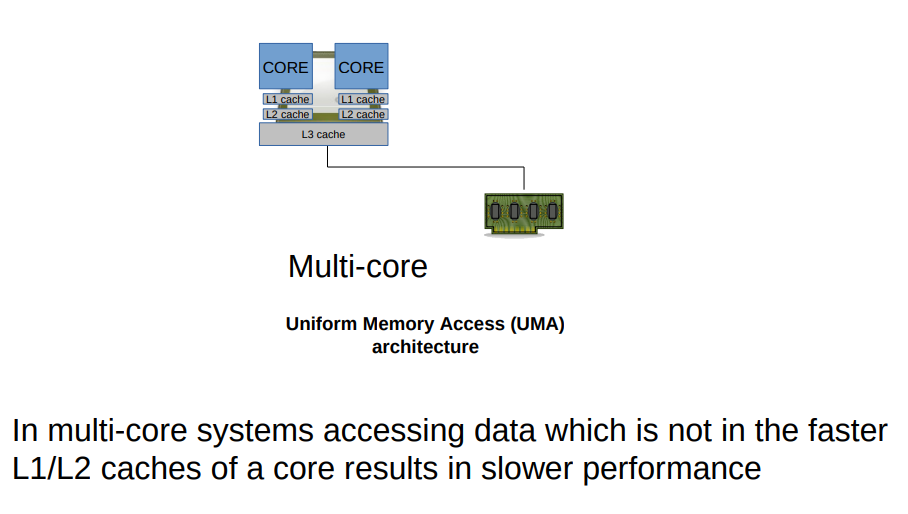
Dati di quelli che sono usati dal thread.

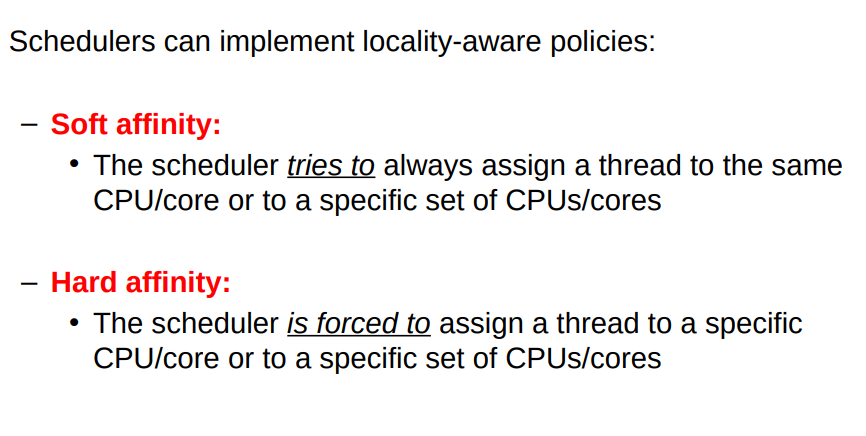
La cache lavora con il principio di località, riducendo la possibilità del rischio di latenza della memoria.

Riduce il numero di dati che devono essere trasferiti dalla memoria principale.

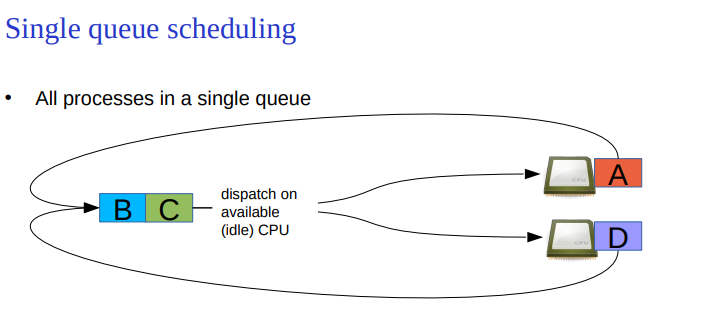
UMA accede ai dati che non sono nella cache locale, NUMA accede ai dati che non sono nella memoria privata.







Hard affinity consente di dare dei core a specifici thread



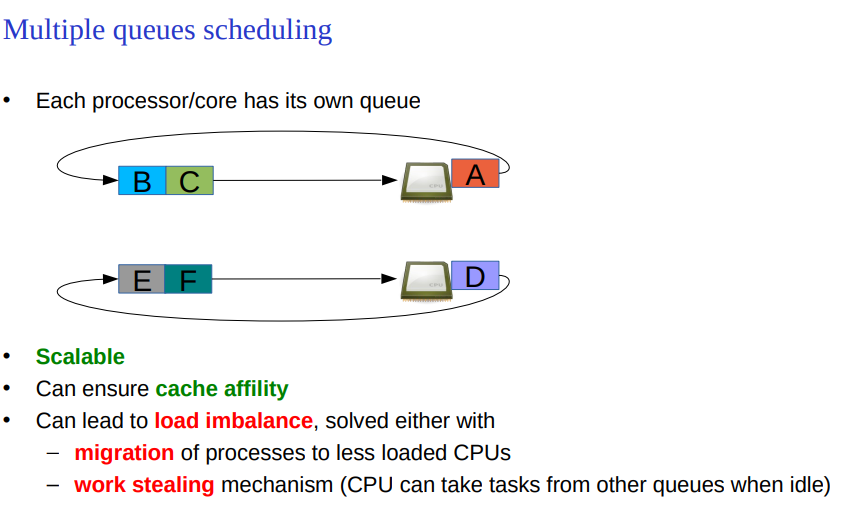
Tutte le cpu vengono sfruttate

Non è un'implementazione scalabile: richiede meccanismi di sincronizzazione per assicurare l’accesso sicuro alla coda condivisa

Problemi di località

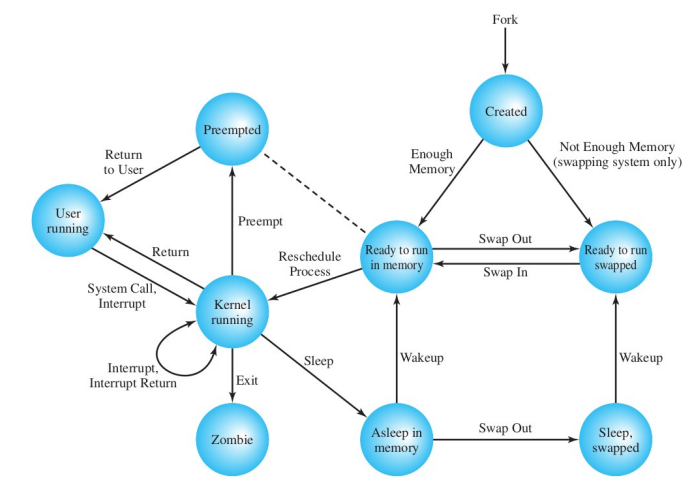
* I processi/threads si muovono tra i vari core
* La cache affinity viene compromessa

Il vantaggio è lo sfruttamento massimo delle risorse

La coda è implementata per ogni core, migliore scalabilità

Lo svantaggio è che le code si svuotano a velocità diverse per ogni core

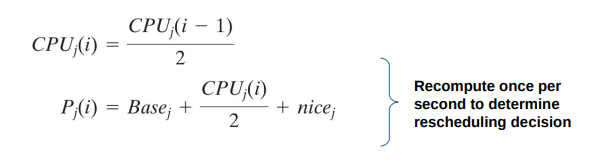
Se il core è in idle può essere preso da task di altre code

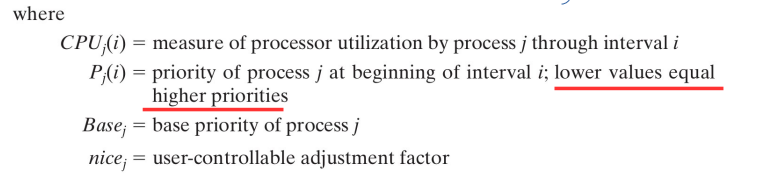


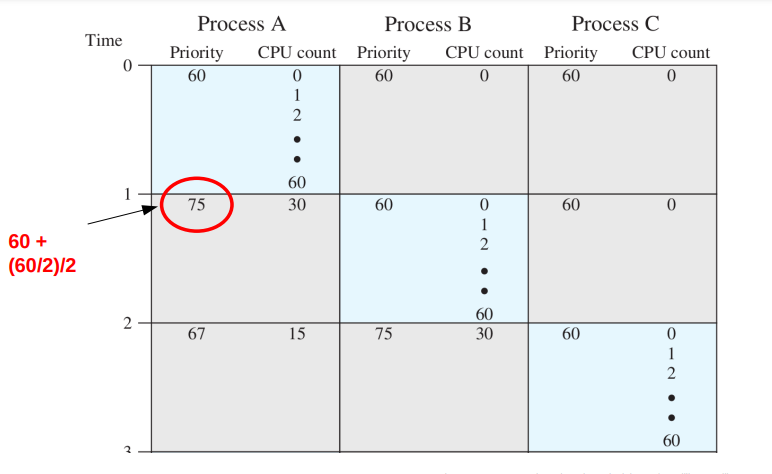
**Scheduling traditional Unix**

Usa un sistema di feedback, che andrà a influenzare lo scheduling

Ogni processo a un priorità base: priorità 1 > priorità 50



Più si usa cpu minore sarà la priorità (valore grande)

Per velocizzare le code usare i bitmaps (dispatch queues in UNIX)

**Linux Scheduler CFS (Completely Fair Scheduler)**  
cerca di assicurare la fairness a tutti i processi

Usa un albero bilanciato, il peso del nodo è il tempo consumato usato sulla cpu, che viene aggiustato dalla priorità.

Su Windows ci sono delle classi di priorità

Per migliorare la responsività alcuni thread ottengono un boost di priorità

