

Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica
Sistemi operativi e reti
A.A. 2022-2023

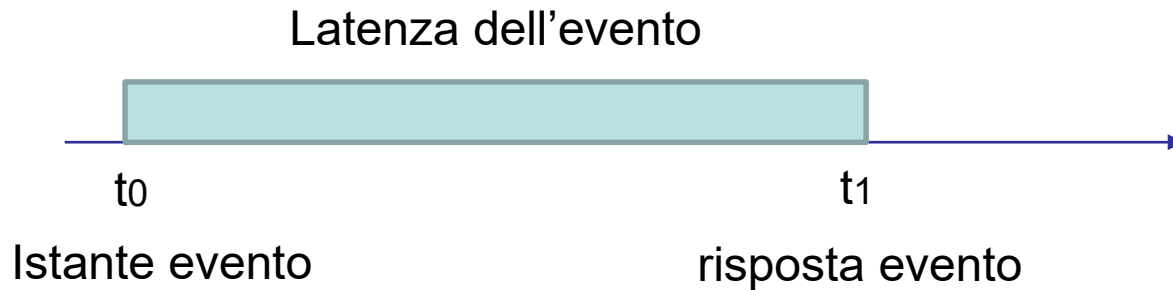
Pietro Frasca

Lezione 14

Martedì 22-11-2022

Algoritmi di scheduling real-time

- La pianificazione della CPU per i sistemi operativi in tempo reale richiede particolari accorgimenti.
- In generale, questi tipi di sistemi si distinguono in sistemi *soft real-time* e *sistemi hard real-time*. I sistemi *soft real-time* garantiscono solo che al processo real-time sarà data la priorità rispetto ai processi non real-time. I sistemi *hard real-time* hanno requisiti più rigidi. Un task deve essere eseguito entro la sua scadenza (deadline).
- In genere, le applicazioni in tempo reale attendono che si verifichi un evento. Gli eventi possono verificarsi sia a livello software, come quando un timer scade, sia nell'hardware, come quando un dispositivo genera un'interruzione.
- Quando si verifica un evento, il sistema deve rispondere il più rapidamente possibile. Definiamo ***latenza dell'evento (event latency)*** l'intervallo di tempo che trascorre da quando si verifica un evento al momento in cui è servito.



Di solito, eventi diversi hanno requisiti di latenza differenti. Ad esempio, la latenza richiesta per il servizio di airbag di un'automobile è molto minore rispetto al servizio del computer di bordo che aggiorna le statistiche sul viaggio, come la velocità media, la distanza percorsa, temperatura dell'acqua del radiatore.

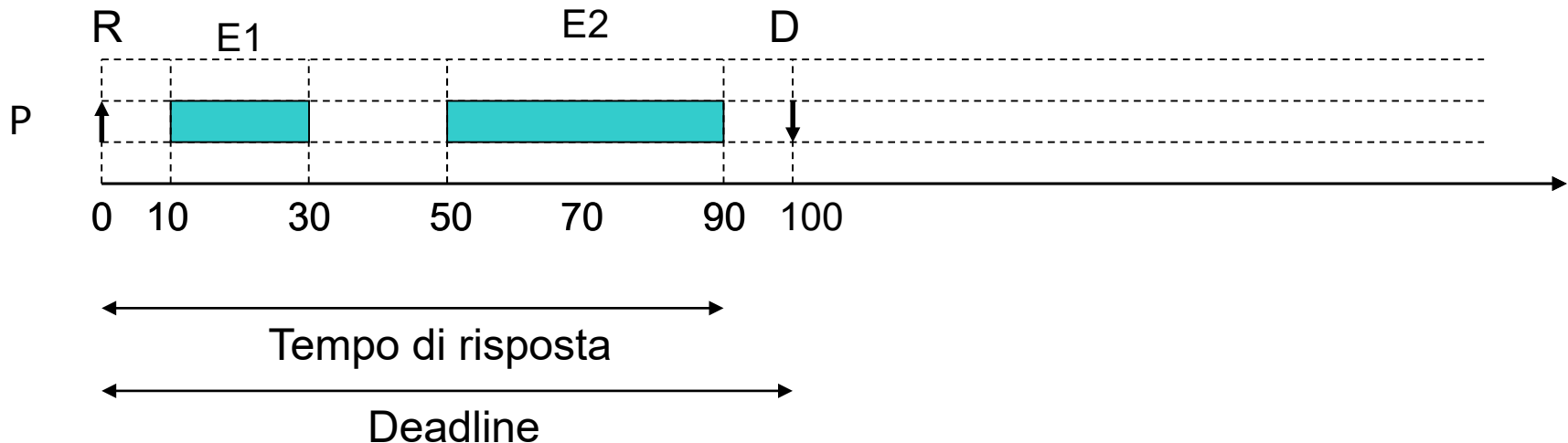
- Lo scheduler di un sistema operativo real-time deve dunque utilizzare un algoritmo basato su priorità con prelazione. Si ricorda che gli algoritmi di pianificazione basati su priorità assegnano a ciascun processo una priorità in base alla sua importanza; compiti più importanti hanno priorità più alte di quelli ritenuti meno importanti.

- Gli algoritmi possono essere statici o dinamici.
- Gli algoritmi statici assegnano le priorità ai processi in base alla conoscenza di alcuni parametri temporali dei processi noti all'inizio. Al contrario gli algoritmi dinamici cambiano la priorità dei processi durante la loro esecuzione.
- I processi real-time fondamentalmente sono caratterizzati dai seguenti parametri:
 - **istante di richiesta**: l'istante in cui il processo entra nella coda di pronto.
 - **deadline**: istante entro il quale il processo deve essere terminato.
 - **tempo di esecuzione**: tempo di CPU necessario al processo per svolgere il suo lavoro.

Esempio

Per il processo in figura si ha:

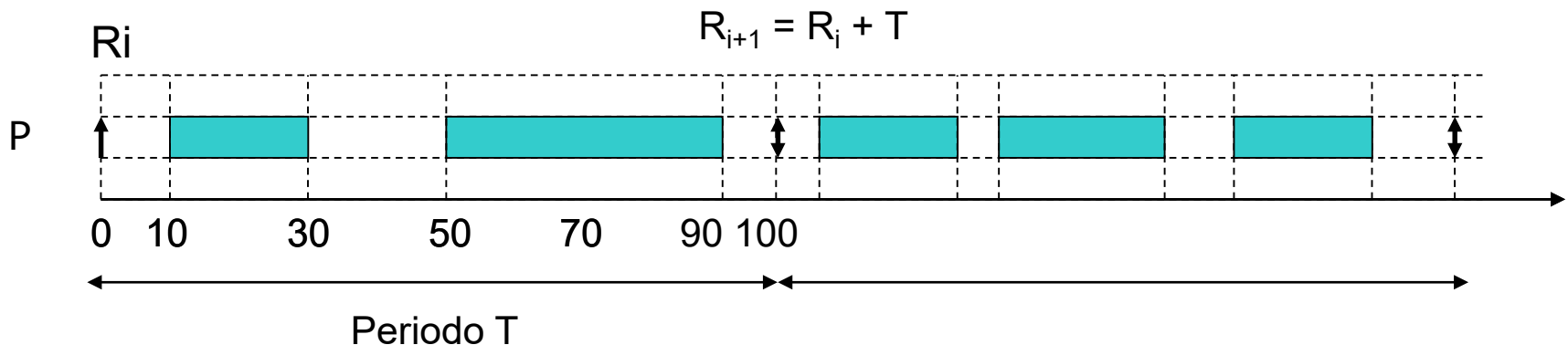
- Istante di richiesta **R** = 0;
- tempo di esecuzione **E** = $E1 + E2 = 20 + 40 = 60$
- deadline **D** = 100
- Tempo di risposta = 90



- I processi real-time possono essere **periodici** o **aperiodici**. I processi periodici vengono attivati ciclicamente a periodo costante che dipende dalla grandezza fisica che il processo deve controllare.
- I processi non periodici vengono avviati in situazioni imprevedibili.
- Consideriamo un algoritmo di scheduling per processi periodici. In tal caso deve essere:

$$R_{i+1} = R_i + T$$

$$T \geq D$$



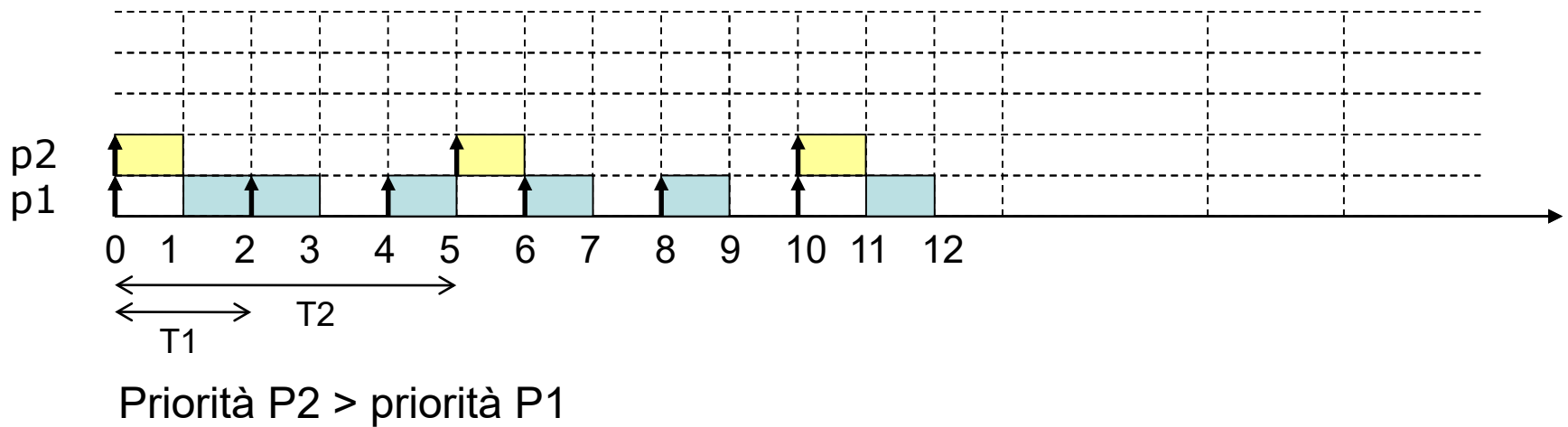
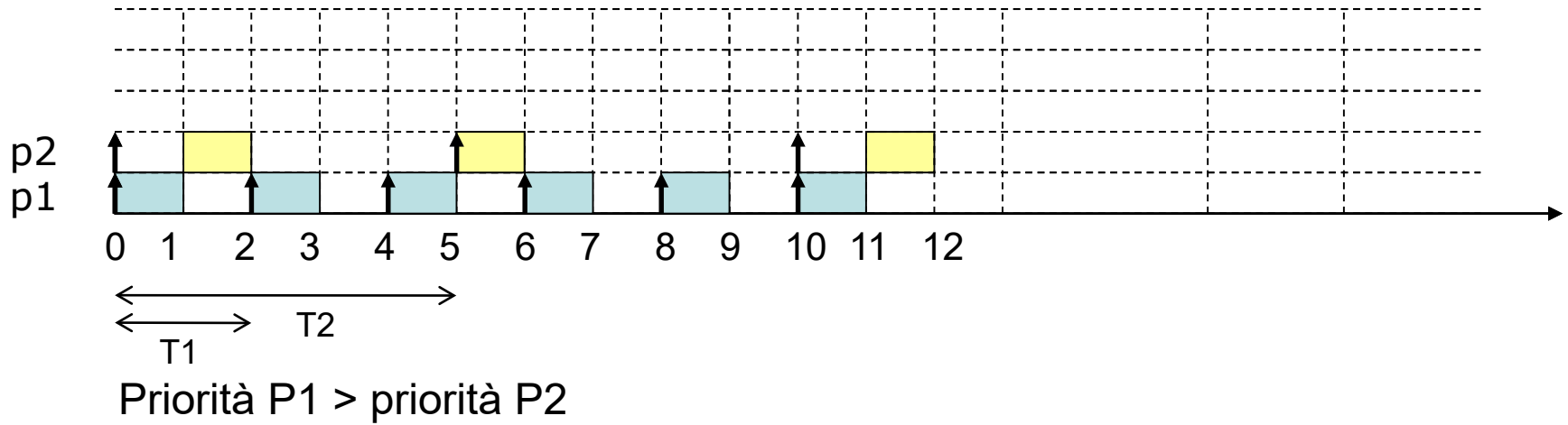
- Dato che in ciascun periodo il tempo di esecuzione di un processo potrebbe cambiare, indichiamo con E_{\max} il tempo massimo di esecuzione di un processo in ciascun periodo T .
- E_{\max} e T sono tempi noti a priori, imposti dall'applicazione real-time.

Rate Monotonic

- Nei SO RT generalmente si utilizzano algoritmi di scheduling con priorità. Il problema è come assegnare la priorità ai processi.
- L'algoritmo ***Rate Monotonic (RM)*** è un algoritmo ottimo, nell'ambito della classe degli algoritmi basati sulle priorità statiche. E' un algoritmo preemptive. Esso assegna la priorità ai processi in base alla durata del loro periodo. In particolare, assegna una priorità maggiore ai processi che hanno periodo minore.
- Per mostrare la validità di tale criterio, consideriamo un esempio in cui due processi P1 e P2 hanno i seguenti parametri temporali:

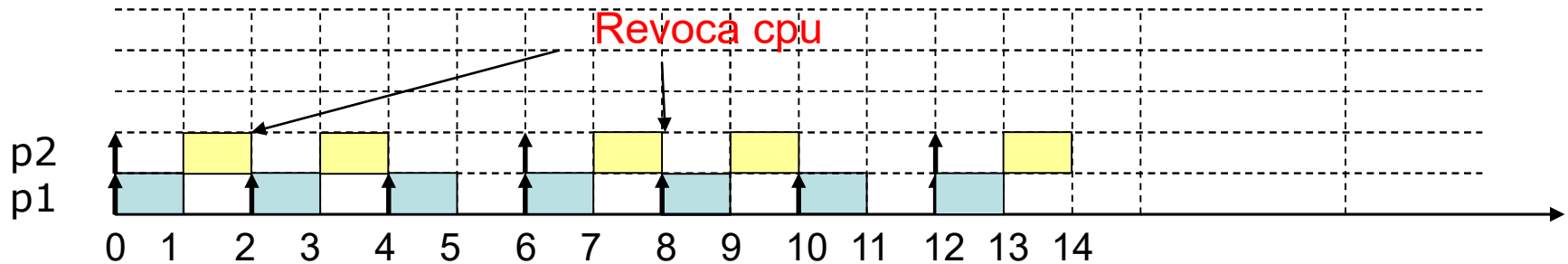
P1: $T1 = 2$; $E1 = 1$

P2: $T2 = 5$; $E2 = 1$

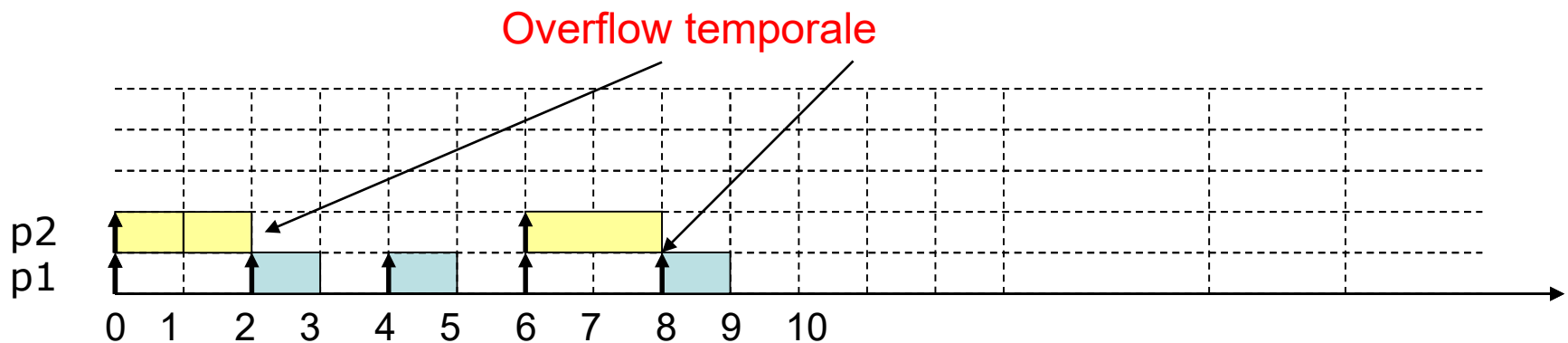


Supponiamo ora che sia $E2 = 2$ $T2=6$, mentre sia ancora $E1 =1$ e $T1 = 2$.

Si ha che nel primo caso i due processi sono ancora schedulabili mentre nel secondo caso no.



Priorità P1 > priorità P2



Priorità P2 > priorità P1

- Tuttavia, anche adottando un criterio ottimo come RM è **necessario** (**ma non sufficiente**) che sia verificata la seguente relazione:

$$U = \sum (E_i/T_i) \leq 1$$

E_i = Tempo di CPU del processo P_i
 T_i = periodo del processo i -esimo per la quale viene ripetuto

Dove **U** è detto **coefficiente di utilizzazione** della CPU.

- E' stato dimostrato, utilizzando **Rate Monotonic**, che affinché un insieme di **N processi** sia schedulabile è sufficiente che il coefficiente di utilizzazione sia:

$$U \leq N(2^{1/N} - 1)$$

Ad esempio per $N=4$ si ha $U=0.76$

per $N=50$ $U=0.698$

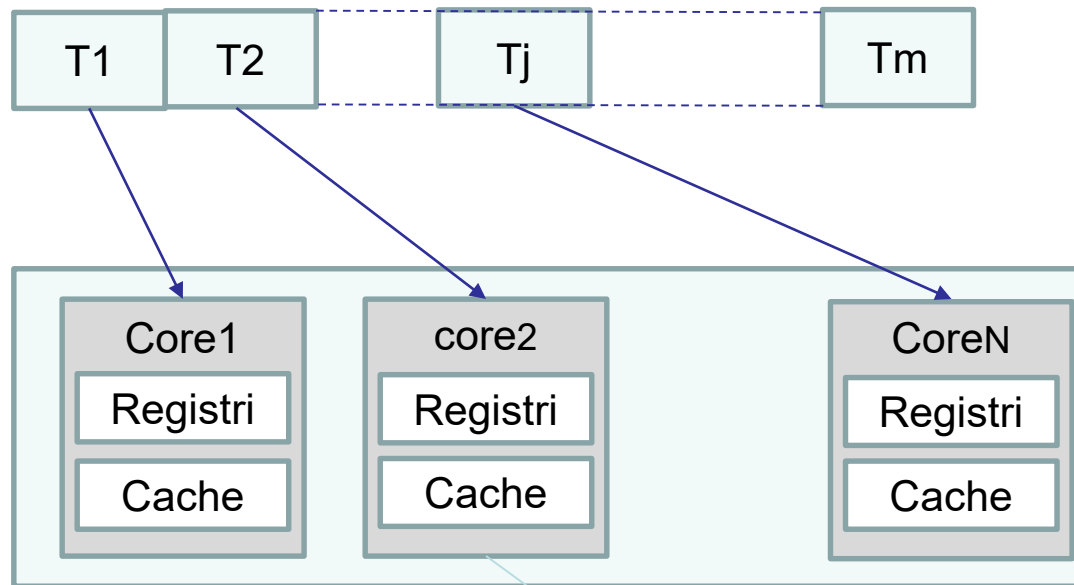
per $N=100$ $U=0.695$

per $N \rightarrow \text{infinito}$ $U \rightarrow \ln 2 = 0,6931471805599453094172$

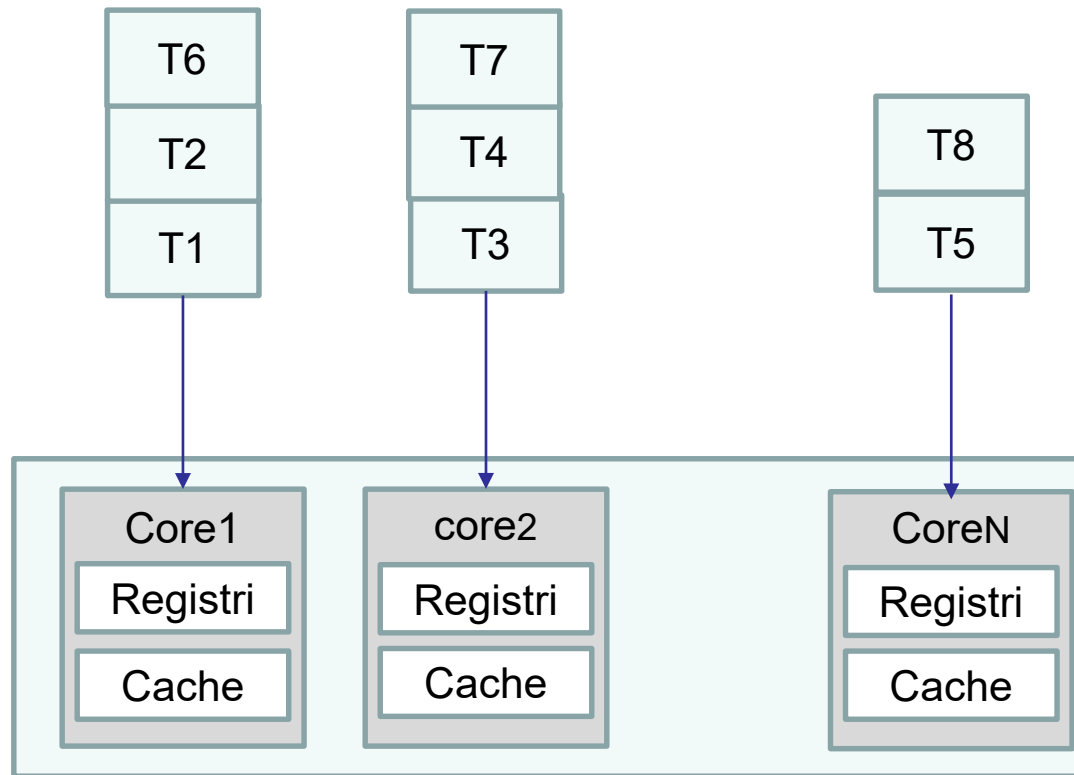
Scheduling nei sistemi multiprocessore

- La nostra discussione fino ad ora si è concentrata sui problemi di pianificazione della CPU in un sistema con un singolo processore. Attualmente, la maggior parte dei computer hanno architetture multiprocessore e pertanto i problemi di pianificazione sono più complessi.
- Nel tempo sono state proposte e sperimentate molte tecniche e come abbiamo visto con la pianificazione della CPU con processore singolo, non esiste una soluzione nettamente migliore di un'altra.
- Un primo approccio alla schedulazione della CPU in un sistema multiprocessore affida tutte le decisioni di pianificazione, l'elaborazione I/O e altre attività di sistema a un singolo processore, detto il **master-server**. Gli altri processori eseguono solo il codice utente. Questo multiprocessing asimmetrico (AMP) è relativamente semplice perché un solo processore accede alle strutture dati del sistema, riducendo la necessità di condivisione dei dati.

- Un secondo approccio utilizza il multiprocessing simmetrico (SMP), in cui ciascun processore esegue tutte le operazioni per la pianificazione. Tutti i processi possono trovarsi in una coda di pronto comune o ogni processore può disporre di una propria coda privata di processi pronti.



- Sui sistemi SMP, è importante mantenere il carico di lavoro bilanciato tra tutti i processori per sfruttare appieno i vantaggi di avere più processori. In caso contrario, uno o più processori potrebbero rimanere inattivi mentre altri processori hanno carichi di lavoro elevati e code di pronto piene.



- Il bilanciamento del carico tenta di mantenere uniformemente il carico di lavoro distribuito tra tutti i processori in un sistema SMP.
- È importante notare che il bilanciamento del carico è in genere necessario solo nei sistemi in cui ogni processore ha una propria coda di pronto privata.
- Sui sistemi con una coda di pronto comune, il bilanciamento del carico è spesso non necessario, perché una volta che un processore diventa inattivo, estrae immediatamente un processo dalla coda comune.
- È anche importante notare, tuttavia, che nella maggior parte dei sistemi operativi attuali che supportano SMP, ciascun processore ha una propria coda di pronto.
- Esistono principalmente due tecniche per il bilanciamento del carico: **push migration** e **pull migration**. Con la migrazione push, un task specifico del SO controlla periodicamente il carico su ciascun processore e, se trova uno sbilanciamento, distribuisce il carico in modo uniforme spostando (push) i processi da processori sovraccarichi a processori inutilizzati o meno occupati. La migrazione pull si verifica quando un processore inattivo estrae un processo dalla coda di pronto di un processore occupato.

- Spesso entrambe le tecniche, push e pull migration, sono implementate in parallelo nei sistemi operativi.
- È da notare che il bilanciamento del carico spesso contrasta i vantaggi dell'affinità del processore

Affinità del processore = capacità di controllare su quale processore/i un thread/processo viene eseguito

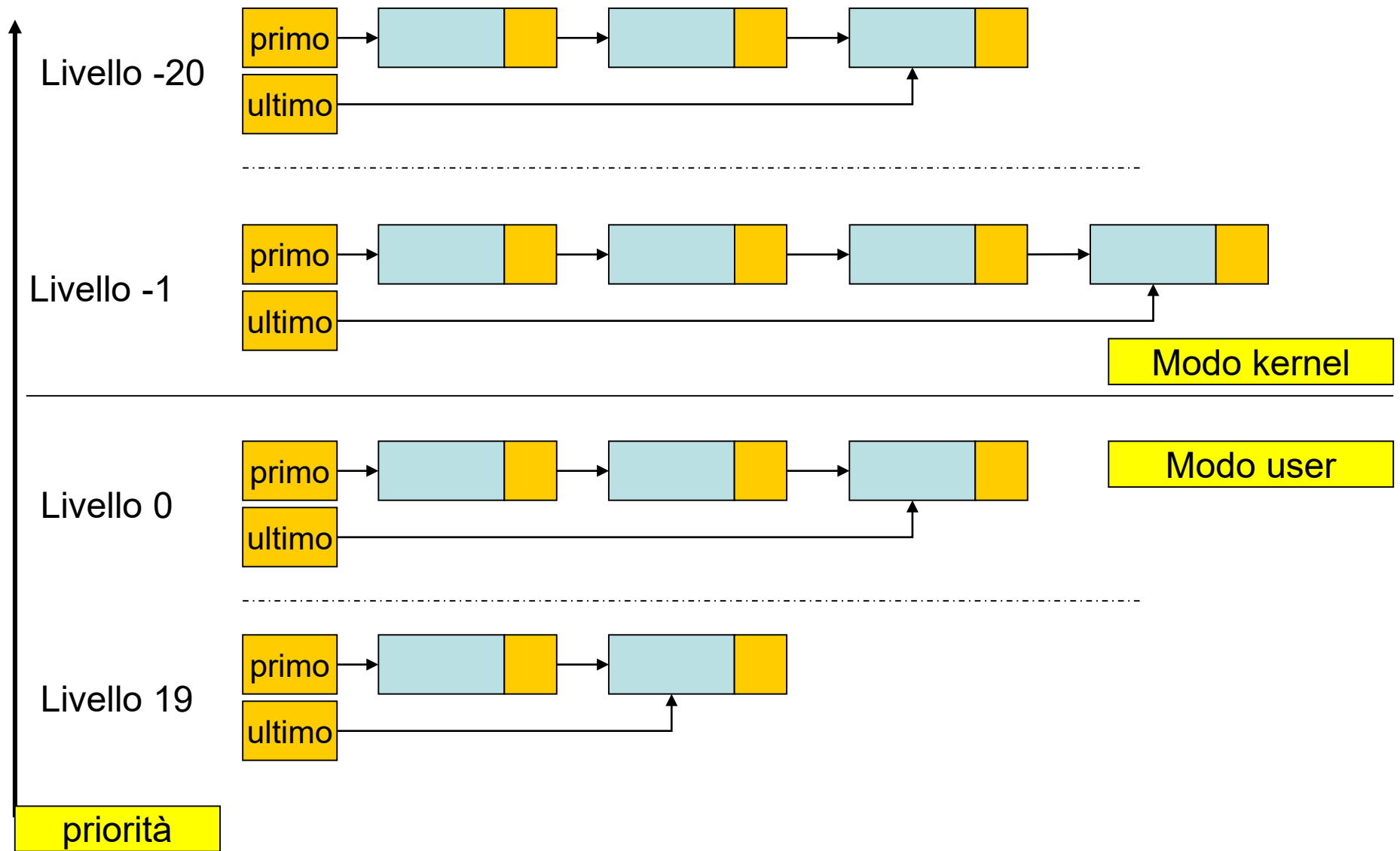
- Quando è in esecuzione, un processo accede ai dati più recenti contenuti nella cache del core cui è stato assegnato. Di conseguenza, gli accessi successivi alla memoria da parte del processo sono spesso soddisfatti nella memoria cache. Se, in un determinato istante, il processo passa a un altro processore, il contenuto della cache del vecchio core deve essere invalidato e la cache del nuovo core deve essere ripopolata.
- A causa dell'elevato costo di invalidare e ripopolare le cache, la maggior parte dei sistemi SMP tenta di evitare la migrazione dei processi da un processore all'altro mantenendo un processo in esecuzione sullo stesso processore. Questo è noto come **affinità del processore**, ovvero un processo ha un'affinità per il processore su cui è attualmente in esecuzione. L'affinità del processore ha diverse realizzazioni.
- Si parla di **affinità soft** quando un sistema operativo tenta di mantenere un processo in esecuzione sullo stesso processore, ma non garantisce che lo farà. Qui, il sistema operativo tenterà di mantenere un processo su un singolo processore, ma è possibile che un processo esegua la migrazione tra i processori.

- Al contrario, alcuni sistemi supportano **l'affinità hard**, consentendo così a un processo di specificare un sottoinsieme di processori su cui può essere eseguito.
- Molti sistemi, come Windows e Linux, offrono affinità sia soft che hard.
- L'architettura della memoria principale di un sistema può influire sui problemi di affinità del processore.

Scheduling in UNIX

- Poiché UNIX è un sistema multiutente e multitasking, l'algoritmo di scheduling della CPU è stato progettato per fornire buoni tempi di risposta ai **processi interattivi**.
- I thread sono generalmente a livello di kernel, pertanto lo scheduler si basa sui thread e non sui processi.
- E' un algoritmo a **due livelli** di scheduler.
- Lo **scheduler a breve termine** sceglie dalla coda dei processi pronti il prossimo processo/thread da eseguire. Utilizza code di priorità, che vengono gestite in modalità RR e la durata del quanto di tempo si aggira tra 20-100ms
- Lo **scheduler a medio termine (swapper)** sposta pagine di processi tra la memoria e il disco (area swap o file di paging) in modo che tutti i processi abbiano la possibilità di essere eseguiti.
- Ogni versione di UNIX ha uno scheduler a breve termine leggermente diverso, ma tutti seguono uno schema di funzionamento basato su code di priorità.

- In Unix le priorità dei processi eseguiti in **modalità utente** sono espresse con valori interi positivi mentre le priorità dei processi eseguiti in **modalità kernel** (che eseguono le chiamate di sistema) sono espresse con valori interi negativi.
- I valori **negativi** rappresentano **priorità maggiori**, rispetto ai valori positivi che hanno priorità minore.
- Lo scheduler a breve termine sceglie un processo dalla coda con priorità più alta. Le code sono gestite in modalità **RR**. Il quanto di tempo assegnato al processo per l'esecuzione dura generalmente **20-100 ms**.
- Un processo è posto in fondo alla coda, quando termina il suo quanto di tempo.



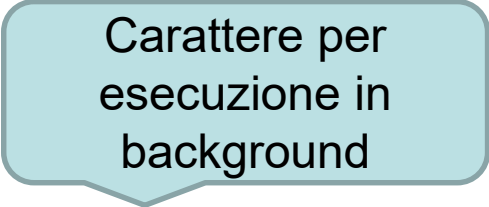
scheduling in unix

- I valori delle **priorità sono dinamici** e sono ricalcolati **ogni secondo** in base ad una relazione che dipende dai seguenti parametri:
 - **valore iniziale (base)**
 - **uso_cpu** uso medio del processore nell'ultima sua esecuzione
 - **nice**

priorità = f(base, uso_cpu, nice)

- Ogni processo è poi inserito in una coda, come mostrato nella figura precedente, in base alla nuova priorità.
- **Uso_cpu** rappresenta, l'uso medio della cpu da parte del processo durante gli ultimi secondi precedenti. Questo parametro è un campo del descrittore del processo (PCB).

- L'incremento del valore del parametro **uso_cpu** provoca lo spostamento del processo in una coda a priorità più bassa.
- Il valore di **uso_cpu** varia nel tempo in base a varie strategie usate nelle varie versioni di UNIX.
- Ogni processo ha, inoltre, un valore del parametro **nice** associato. Il valore di base è 0 e l'intervallo di valori possibili è compreso tra -20 e +19. Ad esempio, con il comando **nice** (che utilizza l'omonima chiamata di sistema **nice**), un utente può assegnare ad un proprio processo un valore *nice* compreso tra 0 e 19. Soltanto il **superuser** (root) può assegnare i valori di *nice* compresi tra -1 e -20 ad un processo.
- Esempio:



Carattere per
esecuzione in
background

nice -n 10 mio_calcolo &

esegue il programma **mio_calcolo** in background assegnando al processo un valore *nice* pari a 10.

- Per quanto riguarda lo scheduling per le estensioni real-time, lo standard P1003.4 di UNIX, cui aderisce anche Linux, prevedono le seguenti classi di thread:
 - **Real-time FIFO;**
 - **Real time round-robin;**
 - **Timesharing**
- I thread real-time FIFO hanno la priorità massima. A questi thread può essere revocata la cpu solo da thread della stessa classe con più alta priorità.
- I thread real-time RR sono simili ai real-time FIFO, ma viene loro revocata la CPU allo scadere del proprio quanto di tempo, il cui valore dipende dalla priorità.
- Le due classi di thread sono **soft real-time**.
- I thread real-time hanno livelli di priorità da 0 a 99, dove 0 è il livello di priorità più alto.

- I thread standard, non real-time, hanno livelli di priorità compresi tra 100 e 139. In totale, quindi si hanno 140 livelli di priorità.
- Il quanto di tempo è misurato in numero di scatti di clock. Lo scatto di clock è detto **jiffy** e dura 1 ms.