

Gestione dei processi nei sistemi operativi Unix/Linux e Windows



Corso di Laurea in Ingegneria Informatica
Università degli Studi di Napoli Federico II
Anno Accademico 2024/2025, Canale San Giovanni



- **Riferimenti**

- P. Ancilotti, M.Boari, A. Ciampolini, G. Lipari, “Sistemi Operativi”, Mc-Graw-Hill (Cap.7, Par. 7.4 – Cap. 8, Par. 8.2)
- www.ostep.org, Cap. 9

- **Approfondimenti**

- W. Stallings, “Operating Systems: Internals and Design Principles” (5th Edition), Prentice Hall (Cap. 3. Cap. 9)
- R. Love, “Linux Kernel Development” (3rd Edition), Cap. 4



Scheduling in Linux

- Il **task** (un flusso di esecuzione) è l'unità fondamentale dello scheduler in Linux
- Viene utilizzato per rappresentare sia per i processi, sia i thread
- Ciascun task è identificato da un **Process ID (PID)**

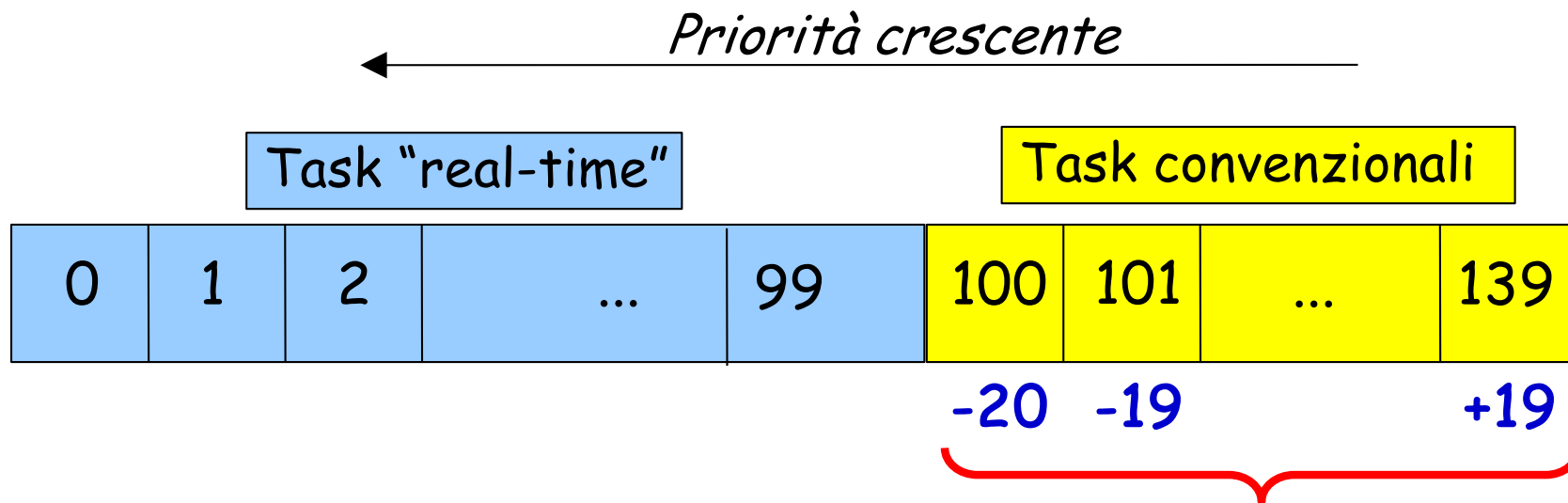


Priorità dei task Linux

- Ad ogni task è attribuita una priorità che ne determina:
 - L'ordine di scheduling
 - Il quanto di tempo assegnato
- Due categorie di task: *real-time* e *convenzionali*



Priorità dei task Linux



- I **programmi utente** sono normalmente in task convenzionali
- La priorità è rappresentata da un intero tra 0 e 39
- L'utente può sommare un valore di correzione (tra -20 e +19) detto **nice value**



Priorità dei task Linux

```
so@so-vbox: ~  
top - 13:39:41 up 2:16, 1 user, load average: 0,03, 0,03, 0,00  
Tasks: 256 total, 1 running, 255 sleeping, 0 stopped, 0 zombie  
%Cpu(s): 1,3 us, 0,3 sy, 0,0 ni, 98,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st  
MiB Mem : 1987,7 total, 139,5 free, 801,4 used, 1046,9 buff/cache  
MiB Swap: 1873,4 total, 1873,4 free, 0,0 used. 1017,2 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1149	so	20	0	625080	304004	118296	S	1,7	14,9	0:20.99	gnome-shell
901	so	20	0	229372	60880	37780	S	0,3	3,0	0:04.74	Xorg
1371	so	20	0	292696	42436	31040	S	0,3	2,1	0:12.24	vmtoolsd
1939	so	20	0	816712	52104	39404	S	0,3	2,6	0:03.13	gnome-terminal-
3434	so	20	0	12000	3996	3212	R	0,3	0,2	0:00.14	top
1	root	20	0	102132	11604	8400	S	0,0	0,6	0:03.03	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.01	kthreadd
3	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	rcu_par_gp
6	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	kworker/0:0H-kblockd
7	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kworker/0:1-events
9	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	mm_percpu_wq
10	root	20	0	0	0	0	S	0,0	0,0	0:00.00	ksoftirqd/0
11	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_sched
12	root	rt	0	0	0	0	S	0,0	0,0	0:00.00	migration/0
13	root	-51	0	0	0	0	S	0,0	0,0	0:00.00	idle_inject/0
14	root	20	0	0	0	0	S	0,0	0,0	0:00.00	cpuhp/0
15	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kdevtmpfs
16	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	netns
17	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_kthre

Nota:

$$PR = 20 + NI$$



Esempio di priorità

```
# Disattiviamo gli altri processori eccetto "cpu0"
$ sudo bash -c 'echo 0 > /sys/devices/system/cpu/cpu1/online'
$ sudo bash -c 'echo 0 > /sys/devices/system/cpu/cpu2/online'
...

# Installa un programma CPU-bound (calcolo numeri primi)
$ sudo apt install mathomatic-primes

# Processo con priorità di default
$ matho-primes 0 9999999999 >/dev/null &

# Processo "favorito" con il comando "nice"
$ sudo nice --10 matho-primes 0 9999999999 > /dev/null &
```



Algoritmi di scheduling in Linux

- Scheduler $O(1)$
- Completely Fair Scheduler (CFS)



Scheduler O(1)

- Introdotto con il kernel Linux 2.6
- Basato sul modello classico **multilevel feedback** usato anche in UNIX
- Obiettivi:
 - Ottenere uno scheduling con **overhead costante** all'aumentare del numero dei processi nel sistema
 - Ottenere un buon compromesso tra tempo di risposta ed equità
 - Utilizzare efficientemente le **architetture SMP**



Scheduler $O(1)$

- Il **tempo impiegato per scegliere** quale processo eseguire è **costante** (fisso), indipendentemente da quanti task ci sono nel sistema
 - fare una scelta fra 10 task in esecuzione...
 - ... impiega lo stesso tempo che per scegliere tra 1000 task in esecuzione

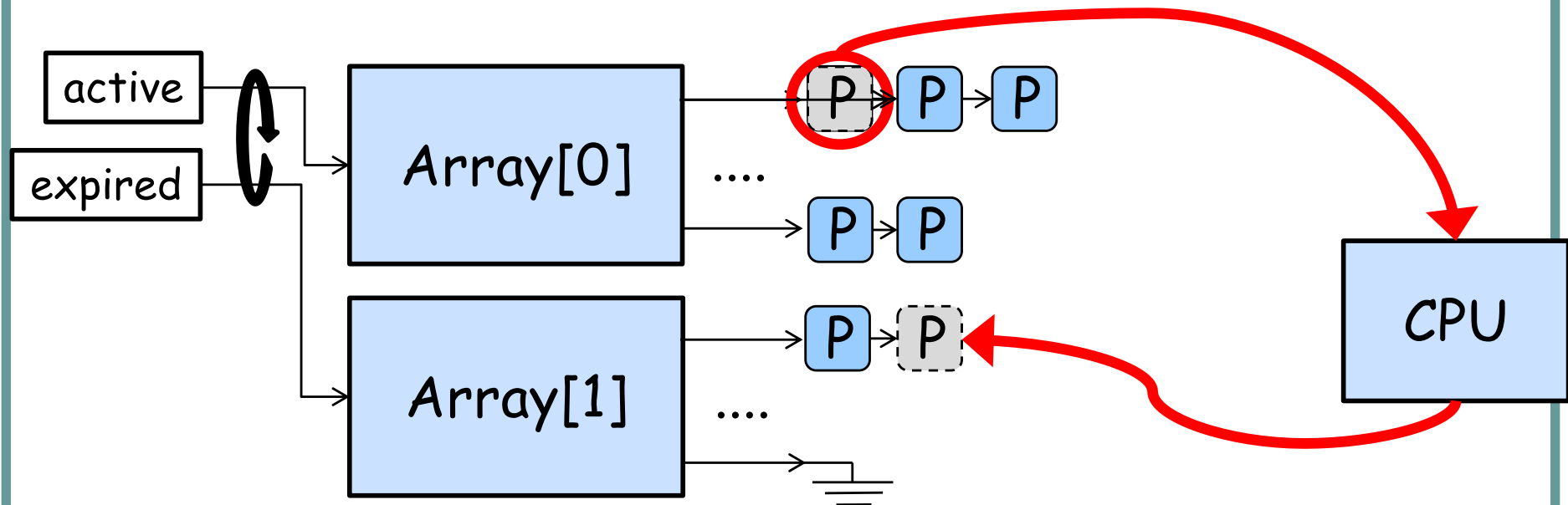


Runqueues

- A ciascun processore è associata una *runqueue*, ossia una struttura dati contenente le *code di processi in attesa*
- Per ridurre il fenomeno della *starvation*, sono introdotte 2 gruppi (array) di code:
 - *Active*: contiene i task che non hanno ancora consumato il quanto di tempo a loro assegnato
 - *Expired*: quando i task hanno eseguito per un intero quanto di tempo, essi vengono spostati nella *coda omologa* di questo gruppo
- Quando tutti i task hanno esaurito il proprio timeslice, si invertono i due gruppi, ed inizia un nuovo round



Array di code di task



Tutte le operazioni (selezione di un processo, ri-accodamento, etc.) hanno una **durata fissa** (si basa su liste linkate)



Priorità statica e dinamica

- Internamente, lo scheduler distingue tra *priorità statica* e *priorità dinamica* dei task
 - *Priorità statica*: coincide con il nice value, determina la durata del **quanto di tempo** assegnato ad un task
 - *Priorità dinamica*: inizialmente pari al nice value, varia durante l'esecuzione; determina la **coda** (runqueue) in cui è inserito (cioè l'**ordine di esecuzione**)



Priorità statica

- Il quanto di tempo (**timeslice**) è proporzionale alla **priorità statica** del task

Tipo di task	Nice value	Timeslice
Creazione	processo padre	metà del padre
Priorità minima	+19	5 ms
Priorità di default	0	100 ms
Priorità massima	-20	800 ms



Calcolo della priorità dinamica

- La **priorità dinamica** determina la **runqueue** in cui è inserito il task
- Si aggiunge/sottrae un **bonus** (fino a +/-5) alla priorità statica
- Identifica e premia i **task I/O bound**

Quando un task passa da **BLOCKED** a **RUNNING**, il suo tempo di attesa viene **aggiunto** a un contatore ("tempo di sleep")

Quando un task passa da **RUNNING** a **BLOCKED**, il suo tempo di esecuzione viene **sottratto** al tempo di sleep

Maggiore il tempo di sleep, maggiore il bonus

CFS scheduler



- Lo scheduler **CFS (Completely Fair Scheduler)** è stato introdotto dalla versione 2.6.23
- Rispetto allo scheduler $O(1)$:
 - Fairness** garantita
 - Non usa euristiche per stimare i task interattivi
 - Rende lo scheduling più facilmente controllabile
 - Supporto al group scheduling
 - Maggiore onere computazionale ($O(\log n)$)



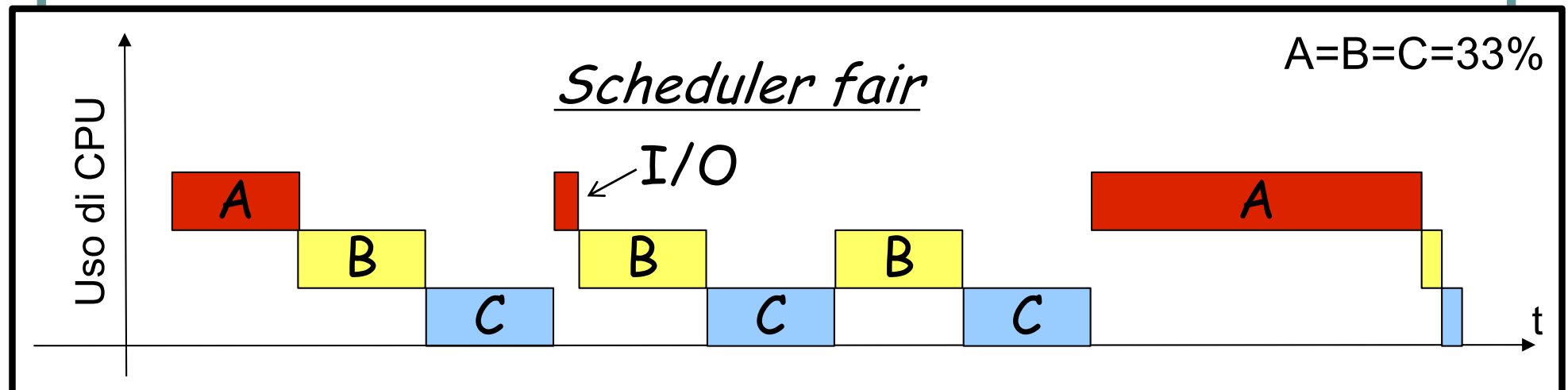
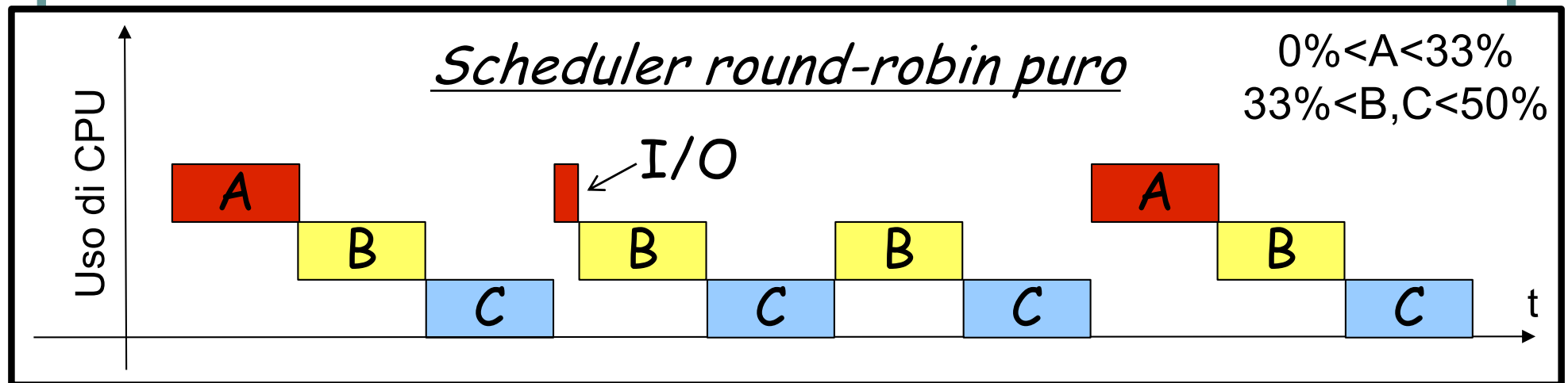
Fair scheduling

- Con gli algoritmi classici (es. con priorità)
 - È difficile controllare la **percentuale di tempo di CPU** concessa ad ogni task
 - Può verificarsi **starvation**
- Nel fair scheduling, a ciascun processo è dedicato una **frazione di tempo proporzionale alla sua priorità**

Fair scheduling



2 processi CPU-bound (**B** e **C**), e 1 processo misto I/O e CPU (**A**),
stessa priorità, 1 processore





CFS: selezione dei task

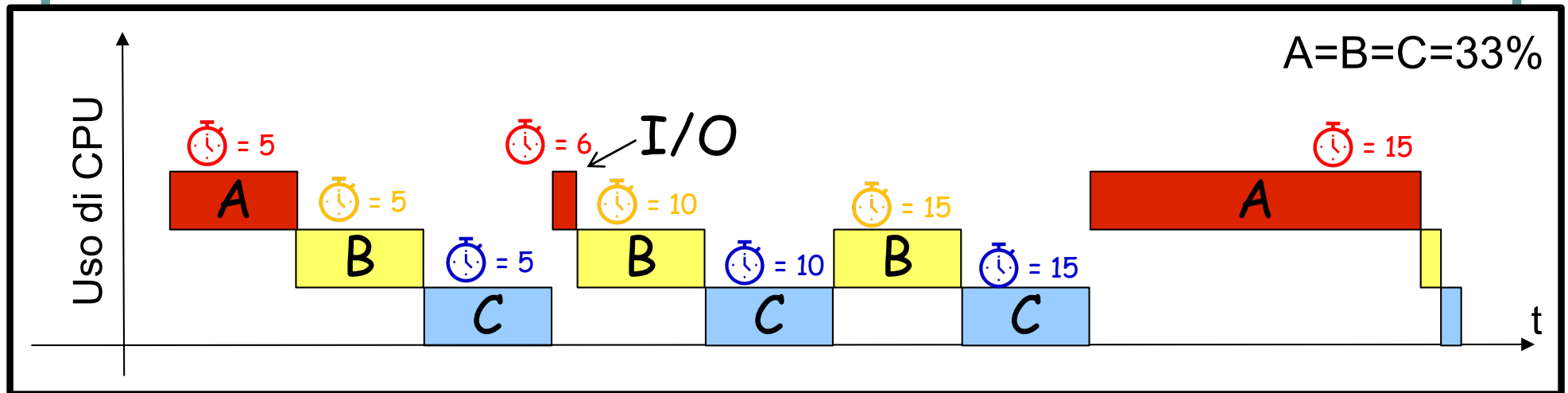
- A ciascun task è associato un "clock virtuale" (*vruntime*)
- Durante l'esecuzione di un task, il suo virtual clock è periodicamente incrementato (tick del timer)

L'algoritmo CFS fa in modo che i clock virtuali dei vari task non differiscano troppo tra loro. CFS seleziona sempre il task con il minor valore di clock virtuale.

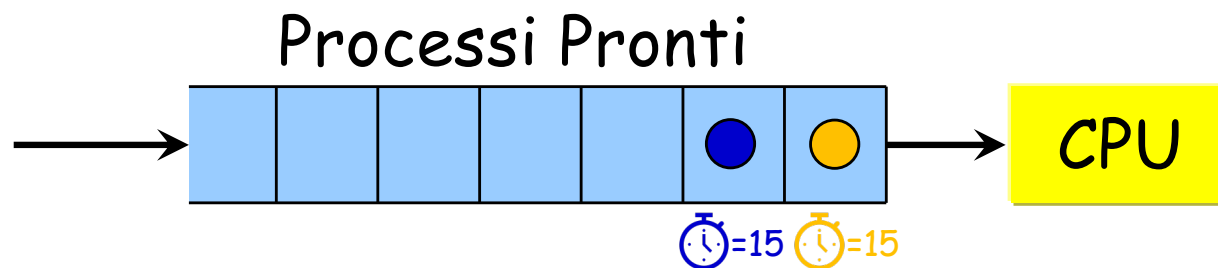


CFS: selezione dei task

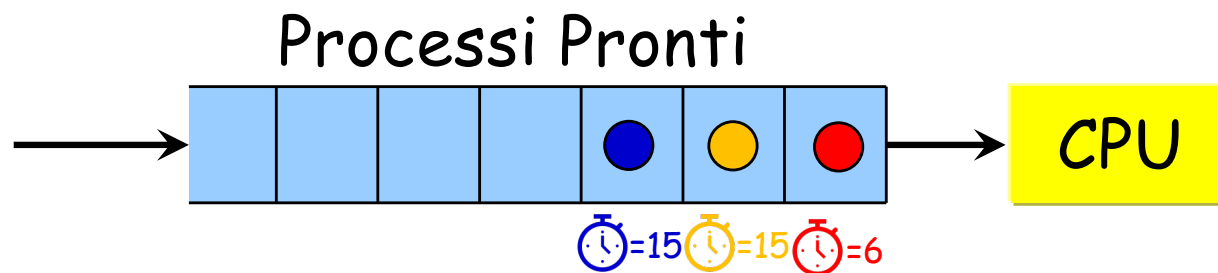
Scheduler fair



CFS: selezione dei task



CFS: selezione dei task

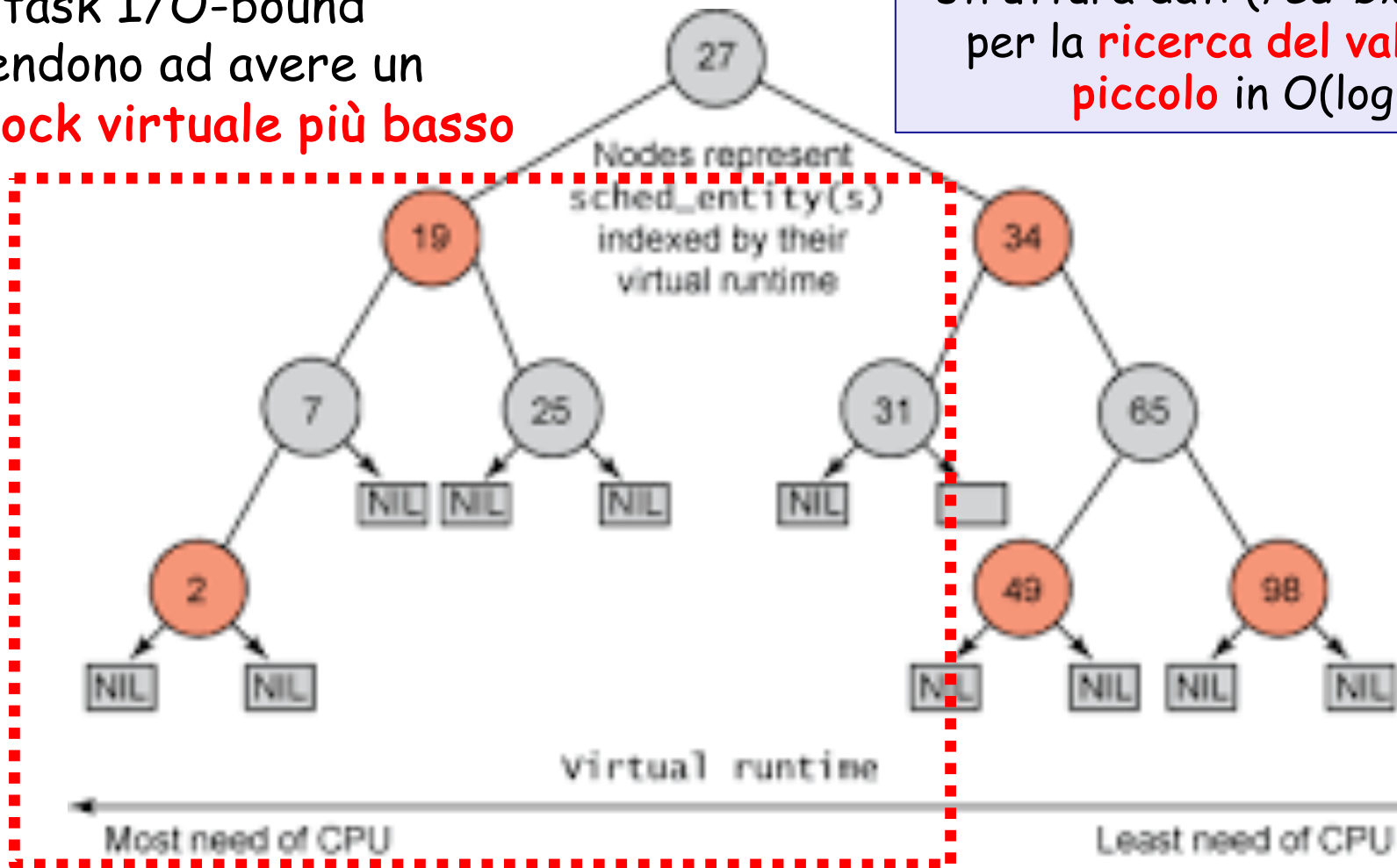




CFS: selezione dei task

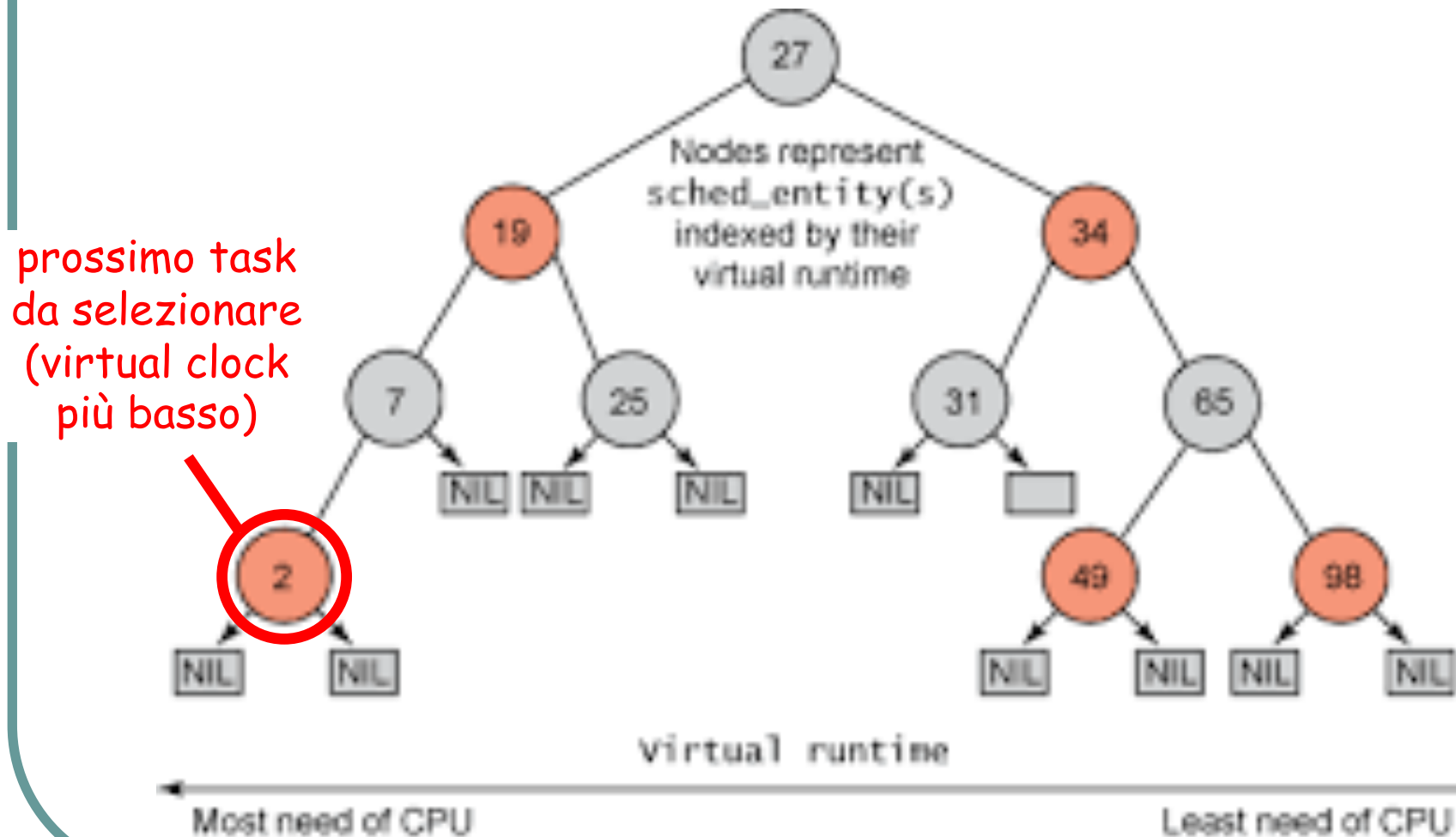
I task I/O-bound tendono ad avere un **clock virtuale più basso**

I task sono raccolti in una struttura dati (*red-black tree*) per la **ricerca del valore più piccolo** in $O(\log n)$





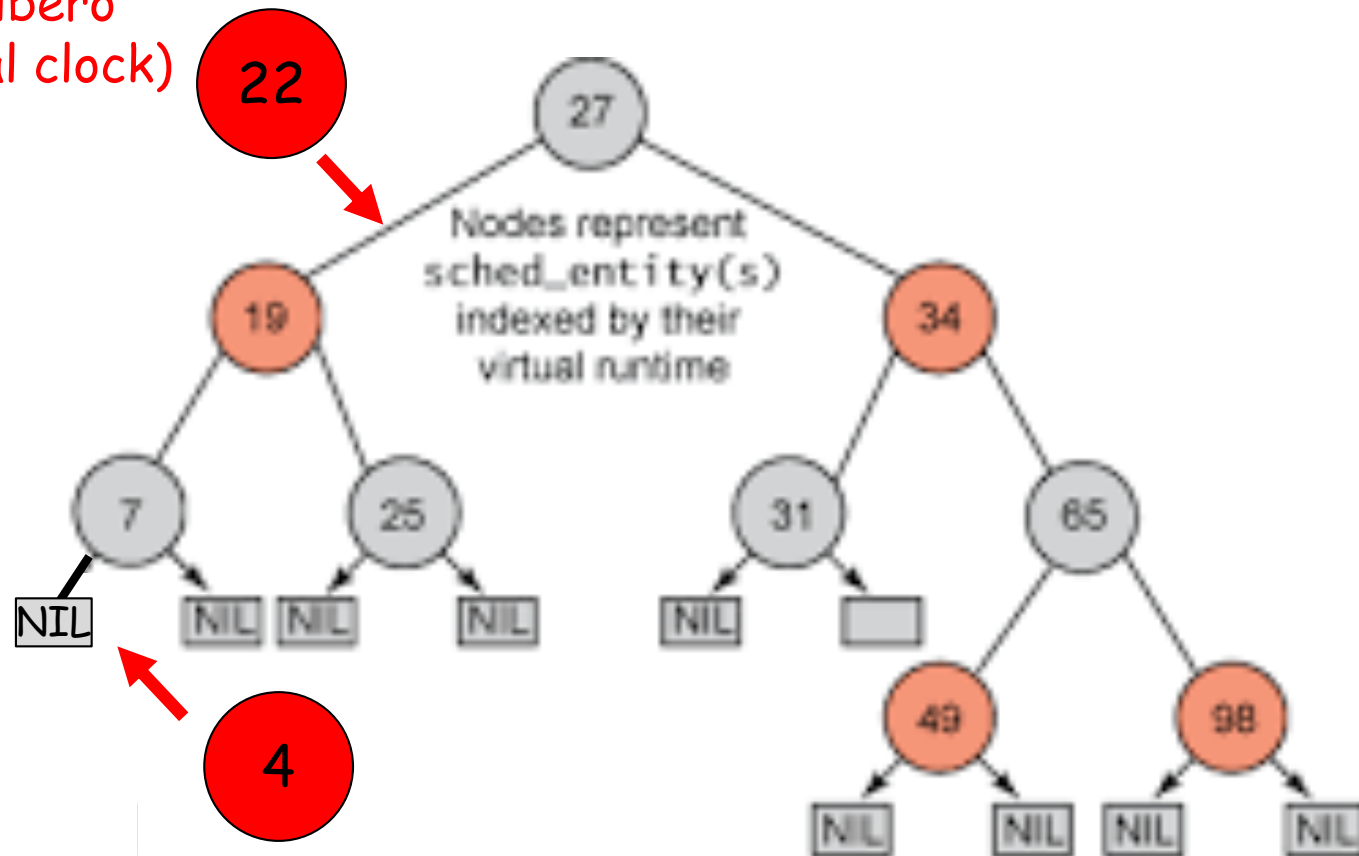
CFS: selezione dei task





CFS: selezione dei task

dopo l'esecuzione, è re-
inserito nell'albero
(in base al virtual clock)



i processi I/O-bound hanno
vruntime basso, tendono a
prelazionare la CPU

Most need of CPU

Least need of CPU



CFS: scelta del quanto di tempo

- Il quanto di tempo non è un valore fissato a priori!
Viene determinato **di volta in volta**
- La **targeted latency** è una finestra temporale (alcuni ms) da **dividere in modo equo** tra i task
- Ogni task **esegue almeno una volta** nella finestra

$$\text{timeslice} = \text{targeted latency} / \text{numero di task}$$



CFS: scelta del quanto di tempo

timeslice = **targeted latency** / **numero di task**

Esempi:

- Targeted latency = 20ms, 4 task
ogni task riceve 5ms di timeslice
- Targeted latency = 20ms, 200 task
ogni task riceve 0.1 ms di timeslice (!!!)

elevato "overhead"
dovuto ai context
switch



CFS: scelta del quanto di tempo

Nel caso i task abbiano "peso" (priorità) diversa:

$$\text{timeslice} = \text{targeted latency} * (\text{peso task} / \text{totale pesi})$$

Esempio:

- Target Latency = 20ms
- Minimum Granularity = 1ms
- Two CPU-Bound Threads
 - Thread *A* has weight 1
 - Thread *B* has weight 4

Time slice for A? 4 ms

Time slice for B? 16 ms



CFS: scelta del quanto di tempo

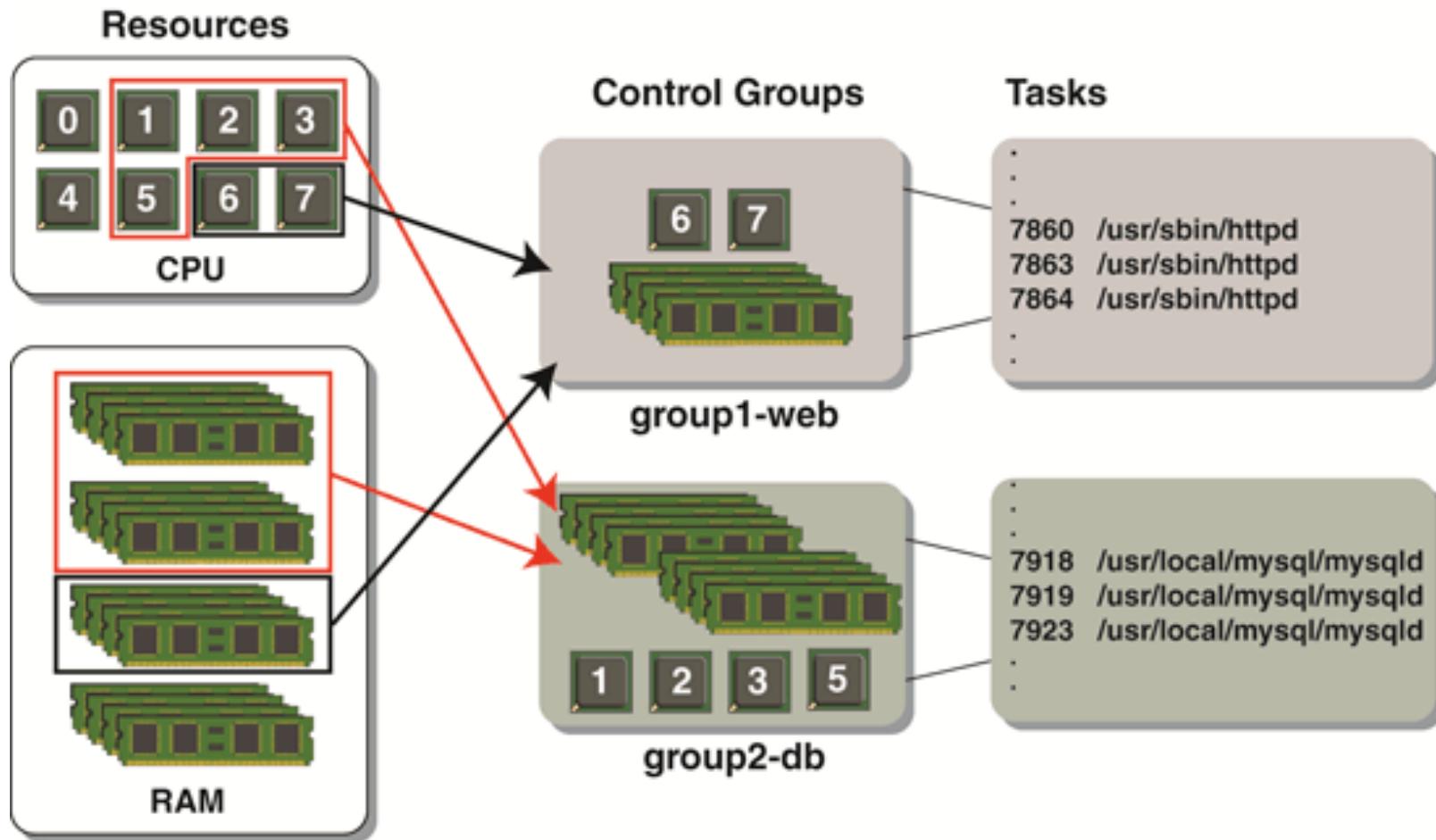
- La targeted latency garantisce un **limite superiore** al tempo di risposta
 - È un parametro configurabile del kernel:
`/sys/kernel/debug/sched/latency_ns`
- Il parametro **minimum granularity** pone un **limite inferiore** al timeslice
 - Es. se minimum granularity = 1ms, anche con 200 task, ognuno riceve comunque 1ms di timeslice
 - È un parametro configurabile del kernel:
`/sys/kernel/debug/sched/min_granularity_ns`

Cgroups



- Il kernel Linux permette di raggruppare più processi in un **control group (cgroup)**, per:
 - **Limitare l'uso di risorse** (CPU, memoria, disk I/O, network, ...) di un gruppo
 - **Prioritizzazione** dell'accesso dei gruppi alle risorse
 - **Tracciabilità (accounting)** dell'uso di risorse
 - **Controllo** dello stato (frozen, stopped, restarted, ...) di tutti i processi in un gruppo, con un solo comando

Cgroups





Cgroups scheduling

- CFS può gestire i task in **gruppi**
- Un gruppo è gestito come una **unica entità schedulabile** in maniera fair
 - La timeslice è assegnata **all'intero gruppo**
 - All'interno di un gruppo, i task si spartiscono il tempo disponibile del gruppo che li contiene



Esempio di cgroups

```
sudo apt-get install cgroup-tools
```

```
sudo cgcreate -g cpu:/cpulimited
```

```
sudo cgcreate -g cpu:/lesscpulimited
```

```
sudo cgset -r cpu.weight=512 cpulimited
```

```
sudo cgset -r cpu.weight=1024 lesscpulimited
```

```
sudo cgexec -g cpu:cpulimited "Prog. P1"
```

```
sudo cgexec -g cpu:lesscpulimited "Prog. P2"
```

Il processo *P1* avrà assegnato la **metà del tempo di CPU** rispetto a *P2*

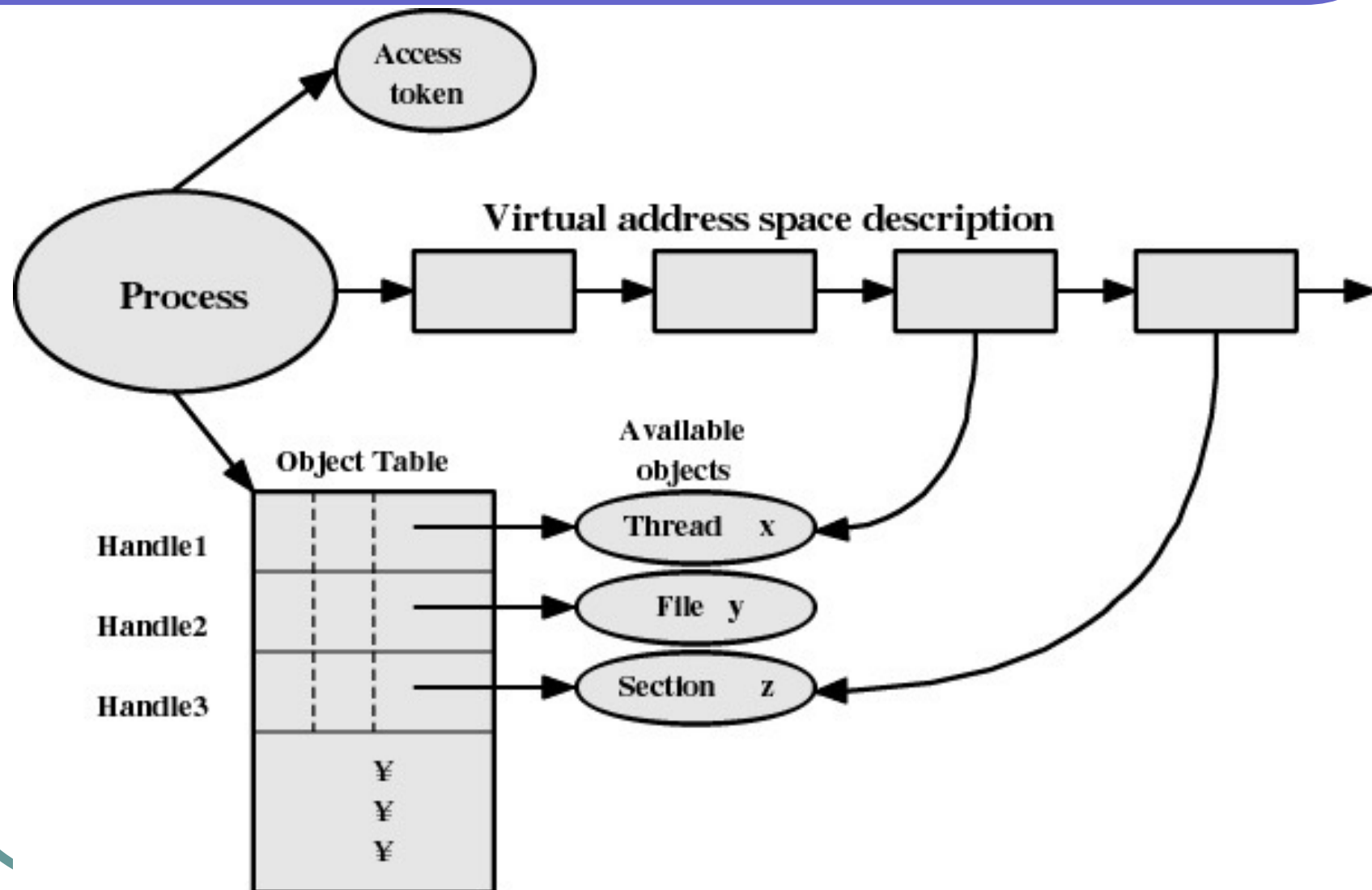


I processi Windows 2000 (W2K)

- W2K è un S.O. multithread, time-sharing.
- Caratteristiche dei processi e thread W2K:
 - I processi/thread W2K sono implementati secondo il paradigma object oriented:
 - Ogni processo è un oggetto (kernel object) che offre stato e funzionalità agli utenti attraverso dei riferimenti (handle)
 - Un processo contiene uno o più thread, che a loro volta sono dei kernel object.
- La concorrenza è realizzata dai thread, per cui un processo W2K deve contenere almeno un thread per eseguire.



Risorse di un processo W2K





Oggetti Process e Thread

Object Type

Process

**Object Body
Attributes**

Process ID
Security Descriptor
Base priority
Default processor affinity
Quota limits
Execution time
I/O counters
VM operation counters
Exception/debugging ports
Exit status

Services

Create process
Open process
Query process information
Set process information
Current process
Terminate process

(a) Process object

Object Type

Thread

**Object Body
Attributes**

Thread ID
Thread context
Dynamic priority
Base priority
Thread processor affinity
Thread execution time
Alert status
Suspension count
Impersonation token
Termination port
Thread exit status

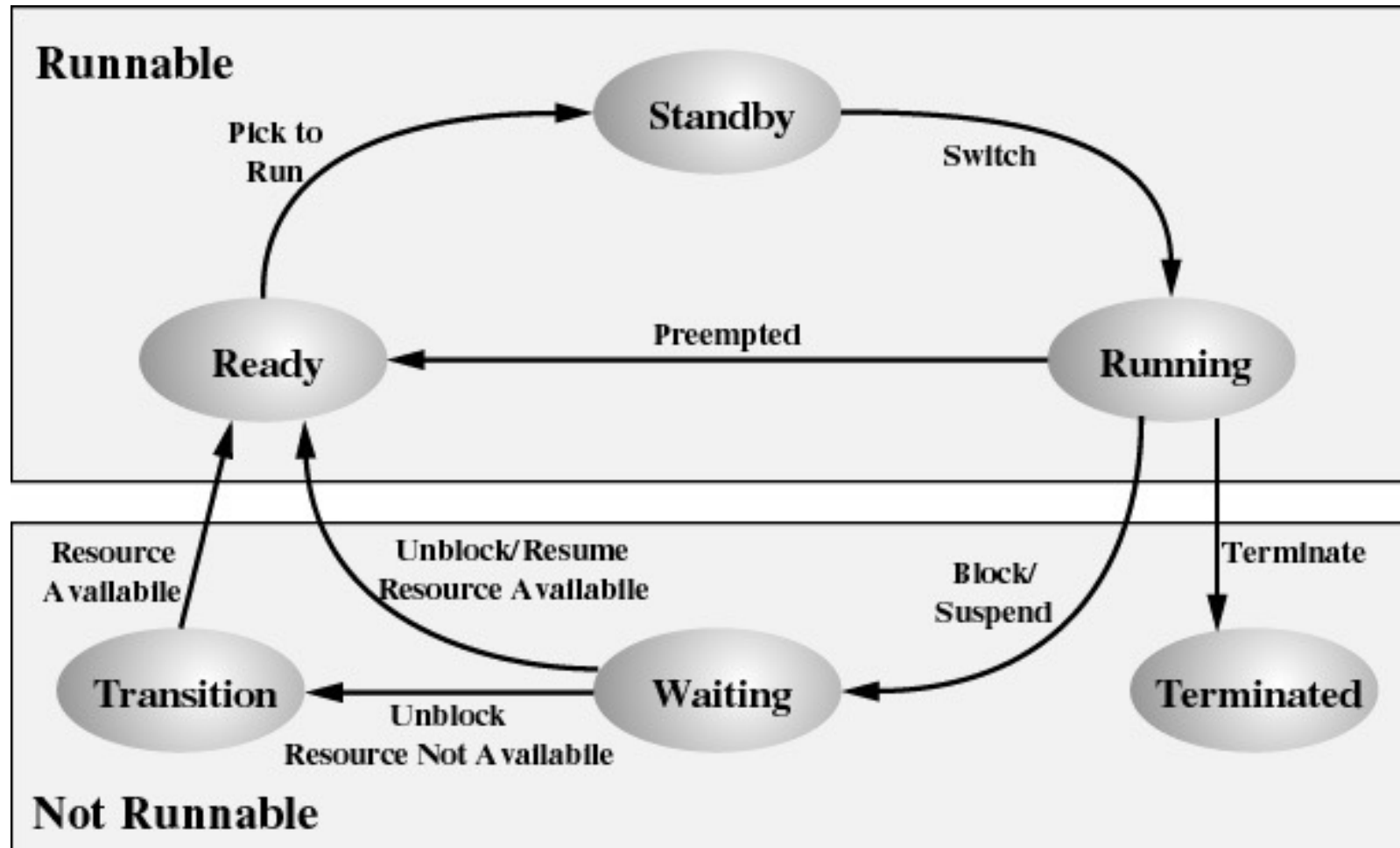
Services

Create thread
Open thread
Query thread information
Set thread information
Current thread
Terminate thread
Get context
Set context
Suspend
Resume
Alert thread
Test thread alert
Register termination port

(b) Thread object



Stati dei Thread in W2K





Scheduling dei Thread in W2K

- Politica di scheduling dei thread: intermedia tra la politica prioritaria e quella “round-robin”
- Ad ogni thread viene associata una priorità assoluta che varia tra 0 e 31
- Viene calcolata come somma di due componenti:
 - Una classe di priorità associata al processo a cui il thread appartiene
 - Una priorità relativa del thread
- Ad ogni classe viene assegnato un valore numerico nominale

Priorità



- Classi di priorità di un processo (in ordine decrescente):
 - **Real-time (24)**: precedenza su ogni thread, da usare con cautela
 - **High (13)**: precedenza sui thread delle classi inferiori, per applicazioni critiche (ad es. Task Manager)
 - **Above Normal (10)**: per i thread leggermente più prioritari del normale
 - **Normal (8)**: maggiormente usato
 - **Below Normal (6)**: per i thread leggermente meno prioritari del normale
 - **Idle (4)**: i thread eseguono quando il sistema non ha altro da fare
- Priorità relative di un thread (in ordine decrescente):
 - **Time Critical**: priorità uguale a 15 (31 se la classe è Real-time)
 - **Highest**: +2 rispetto alla nominale (valore della classe)
 - **Above Normal**: +1 rispetto alla nominale
 - **Normal**: si usa la priorità nominale
 - **Below Normal**: -1 rispetto alla nominale
 - **Lowest**: -2 rispetto alla nominale
 - **Idle**: priorità uguale a 1 (16 se la classe è Real-time)

Priorità



The screenshot shows the Windows Task Manager interface. The 'Processes' tab is active, displaying a list of running processes. The process 'VirtualDub.exe' is selected, and its context menu is open. The 'Set priority' option is highlighted, and a secondary menu is displayed showing the available priority levels. The 'Below normal' priority is currently selected and highlighted in red.

Process Name	PID	State	Session Name	Session ID	Private Bytes
VGAAuthService.exe	2072	Running	SYSTEM	00	16 K
VirtualDub.exe			admin	00	3,628 K
vmacthlp.exe			SYSTEM	00	16 K
vmtoolsd.exe			SYSTEM	00	3,372 K
vmtoolsd.exe					40 K
wininit.exe					0 K
winlogon.exe					76 K
WmiPrvSE.exe					40 K
WUDFHost.exe					56 K

Context menu options for VirtualDub.exe:

- End task
- End process tree
- Set priority** (selected)
- Set affinity
- Analyze wait chain
- UAC virtualization (checked)
- Create dump file
- Open file location

Priority selection menu:

- Realtime
- High
- Above normal
- Normal
- Below normal** (selected)
- Low



Scheduling dei Thread in W2K

- Per **ogni livello di priorità**, lo scheduler mantiene una **coda gestita con Round Robin**, con quanto 10 ms
- Lo scheduler esegue il primo thread della coda a priorità più alta che contiene almeno un thread pronto
- Il thread può eseguire fino a quando:
 - Il quanto temporale termina
 - Il thread si sospende in attesa di un evento esterno
 - Un thread a priorità più alta viene attivato
- In tutti e tre i casi, lo scheduler viene invocato nuovamente per scegliere il successivo thread



Meccanismo del *dynamic priority boost*

- **Dynamic Priority Boost:** permette di ridurre il **tempo di risposta** dei **thread interattivi**
 - All'**attivazione** di un thread, la sua priorità base viene aumentata o diminuita fino a 2 punti
 - Al **consumo completo di un quanto di tempo**, la sua priorità viene decrementata di un punto
 - Se il thread si **sospende** in attesa di un evento di I/O, la sua priorità viene incrementata
 - In ogni caso la priorità del thread non può essere inferiore alla sua priorità base



Meccanismo del *dynamic priority boost*

