SMP e Scheduling Multiprocessore

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

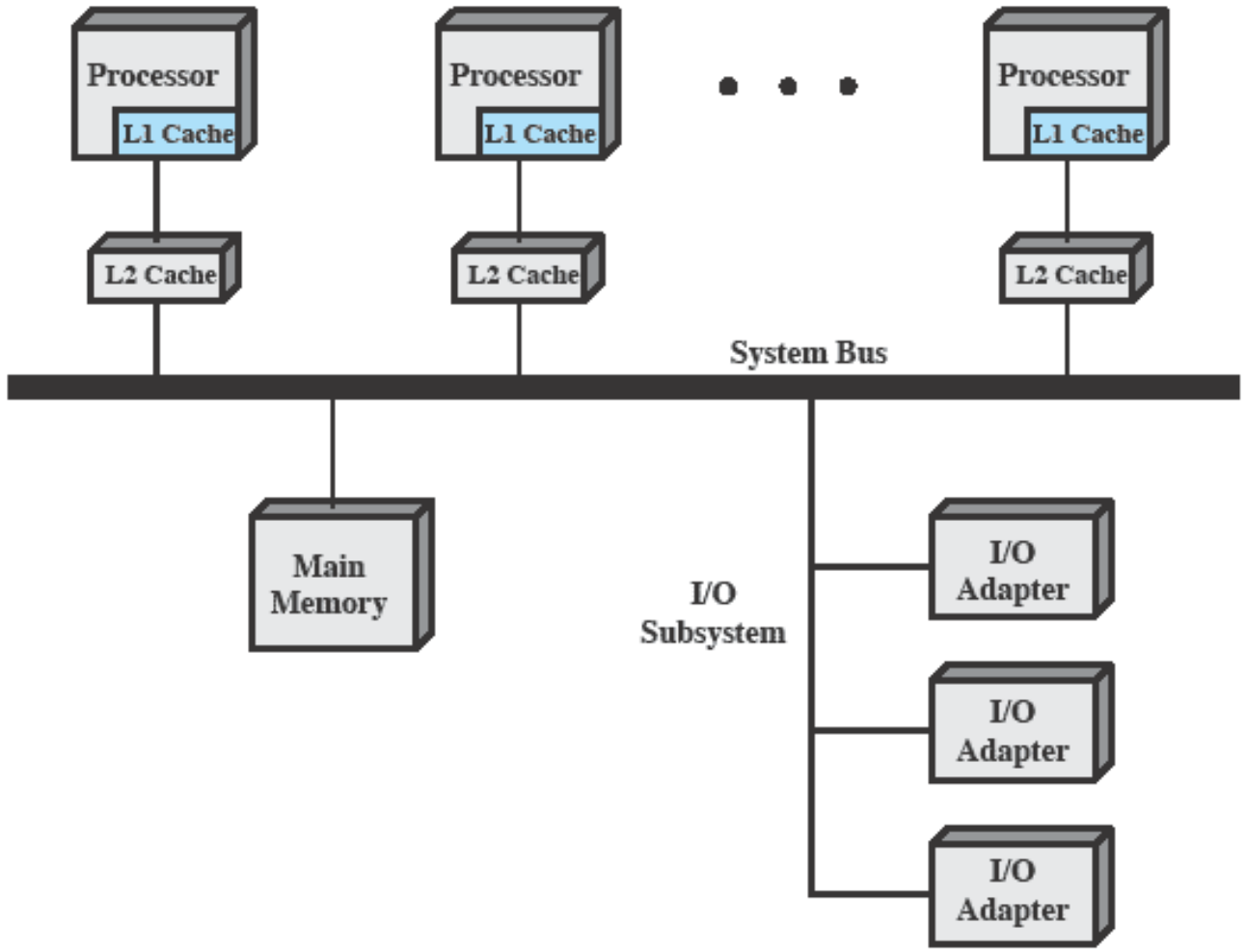
Sommario della lezione

- Sommario della lezione:
  - Cenni su architetture SMP
  - Scheduling multiprocessore
    - Problematiche di progettazione
    - Dispatching: load sharing, gang scheduling
- Riferimenti
  - P. Ancilotti, M.Boari, A. Ciampolini, G. Lipari, "Sistemi Operativi", Mc-Graw-Hill (par. 2.10)
  - www.ostep.org, Cap. 10
  - Stallings, "Operating Systems" 6th ed., par. 4.1 e 10.1

Tecnologie multi-processore

- **Symmetric Multi-Processing (SMP)**: più CPU identiche (una per chip) sono collegate a una stessa memoria condivisa, e hanno pieno accesso ai dispositivi di I/O
- **Multi-core CPU**: più CPU ("core") sono sullo stesso chip
- **Hyperthreading**: uno stesso core ha risorse multiple (ALU, registri, etc.) e può eseguire più programmi contemporaneamente

Organizzazione tipica di un’architettura SMP



Esempio di multi-core (Intel Core i7-5960X)

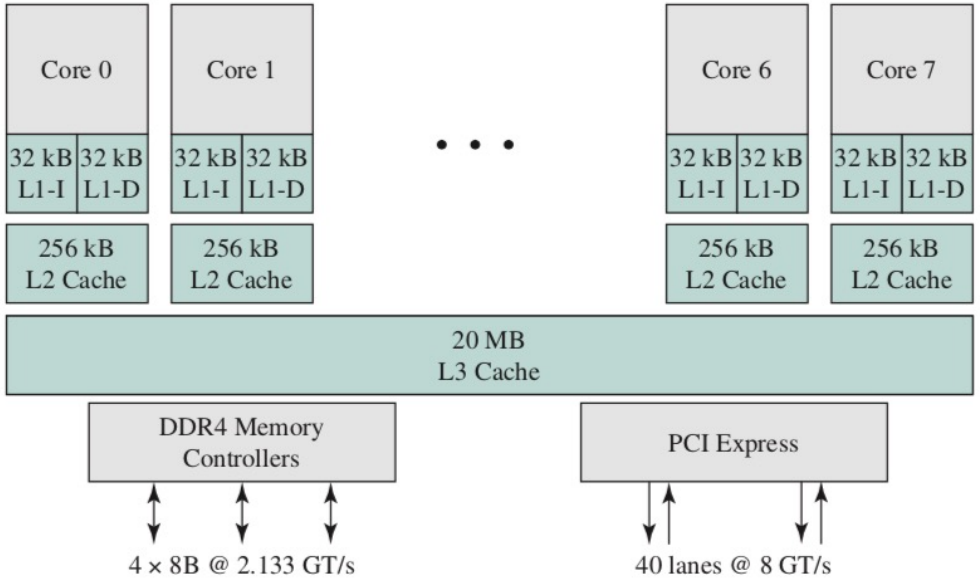
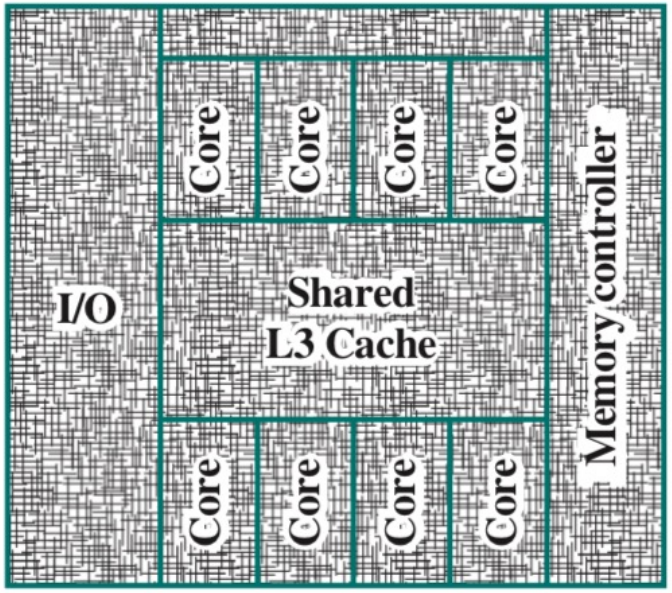


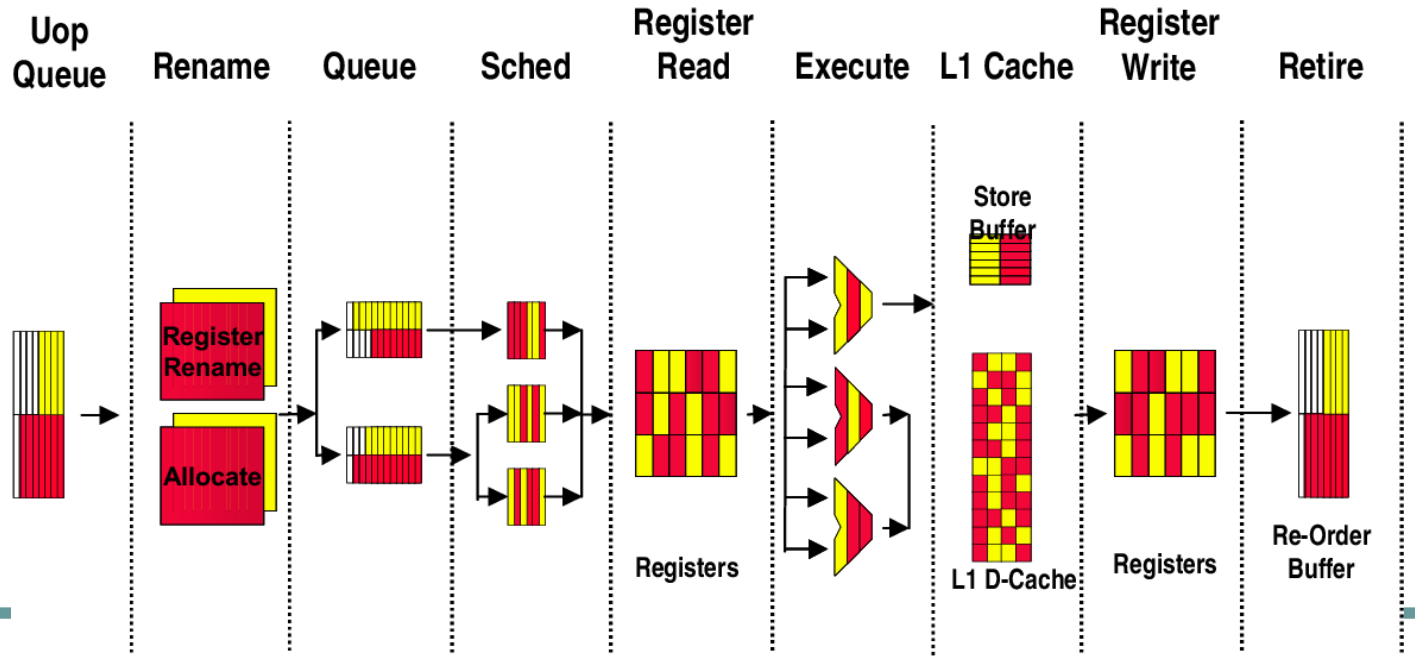
Diagramma a blocchi



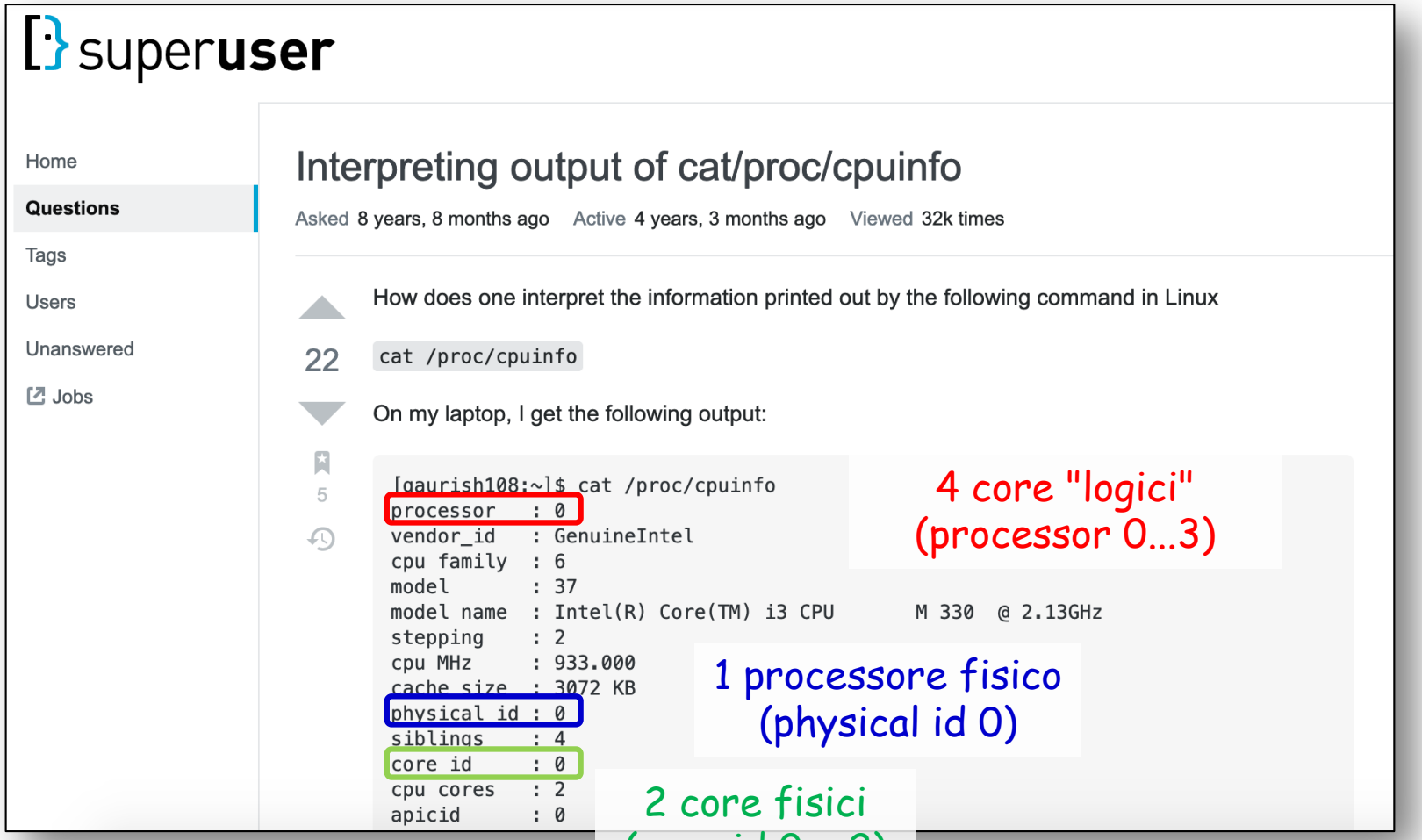
Layout sul chip fisico

Hyperthreading

- Il SO "vede" più **core virtuali**
- Ogni core virtuale esegue un programma (come una vera CPU)
- I core virtuali eseguono in realtà sullo **stesso core fisico**
- Il core fisico **condivide i componenti hardware** (es. ALU) tra i core virtuali



/proc/cpuinfo



superuser

Home
Questions
Tags
Users
Unanswered
Jobs

Interpreting output of cat/proc/cpuinfo

Asked 8 years, 8 months ago Active 4 years, 3 months ago Viewed 32k times

How does one interpret the information printed out by the following command in Linux

22

`cat /proc/cpuinfo`

On my laptop, I get the following output:

5

```
[gaurish108:~]$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 37
model name  : Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
stepping    : 2
cpu MHz     : 933.000
cache size  : 3072 KB
physical id : 0
siblings    : 4
core id     : 0
cpu cores   : 2
apicid      : 0
```

4 core "logici" (processor 0...3)

1 processore fisico (physical id 0)

2 core fisici (core id 0 e 2)

https://superuser.com/questions/388115/interpreting-output-of-cat-proc-cpuinfo

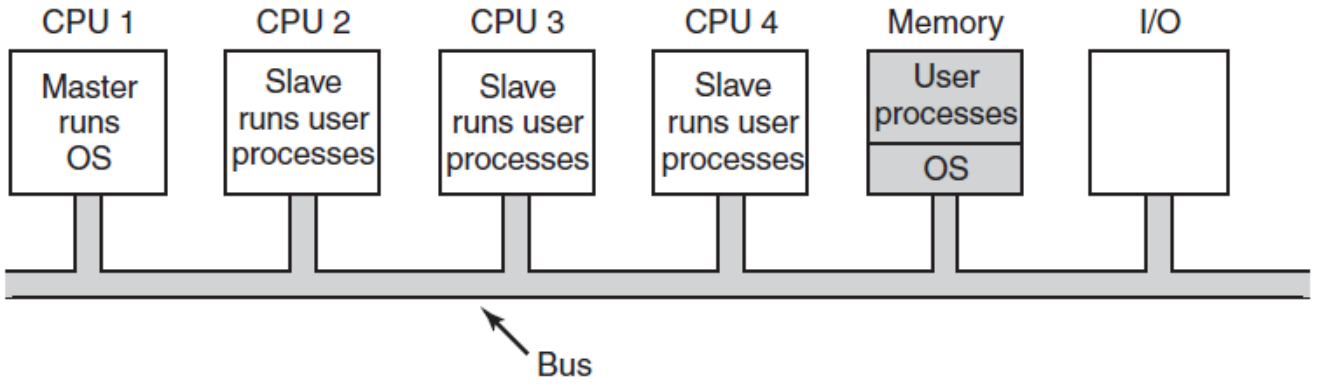
SO per SMP

- Scelte di design:
  - Esecuzione del meccanismo di assegnazione
  - Assegnazione dei processi ai processori

Dove esegue il meccanismo di assegnazione? MASTER/SLAVE

- Approccio master/slave: il kernel esegue su un solo processore, detto **master**, responsabile dello scheduling
- Gli altri processori (**slave**) possono eseguire solo processi utente
- Gli slave inoltrano le system call fatte dai processi al master

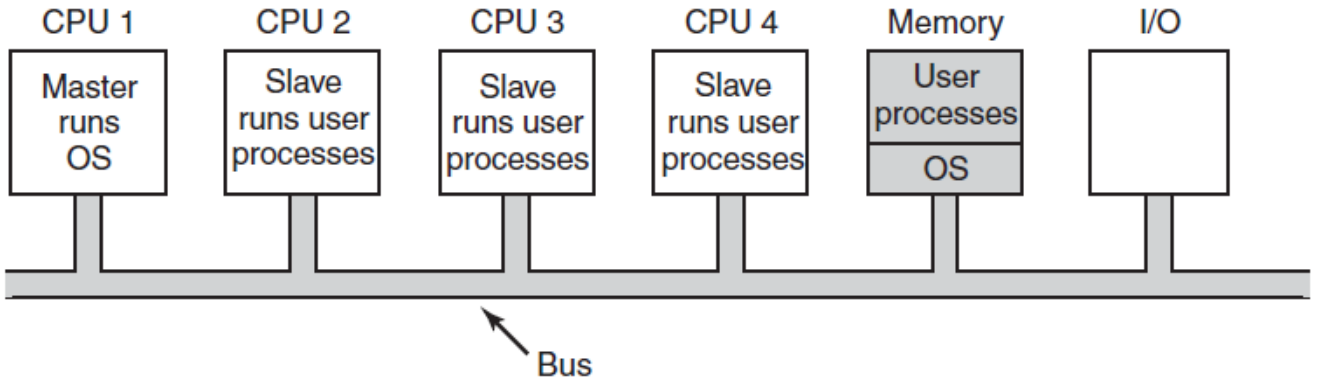
Approccio master-slave



Dove esegue il meccanismo di assegnazione? MASTER/SLAVE

- Vantaggi: approccio di semplice implementazione
- Svantaggi:
  - il master costituisce un *single point of failure*
  - il master è un **collo di bottiglia** per le prestazioni

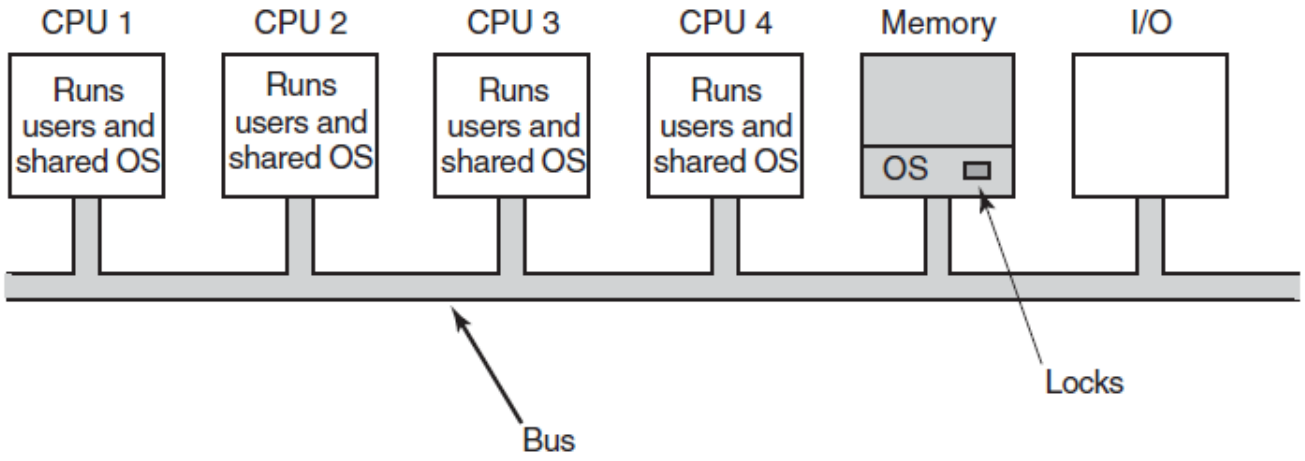
Approccio master-slave



Dove esegue il meccanismo di assegnazione? APPROCCIO PEER

- Approccio peer: il kernel può eseguire su **tutti i processori**, anche contemporaneamente
- Ogni processore gestisce autonomamente lo scheduling

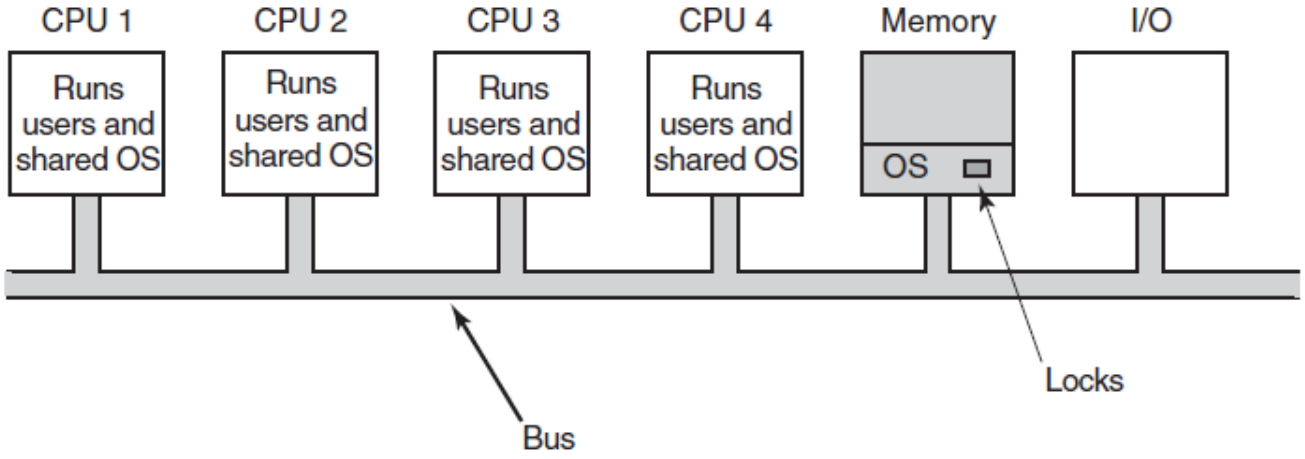
Approccio peer



Dove esegue il meccanismo di assegnazione? APPROCCIO PEER

- Approccio di più complessa implementazione, in quanto richiede la **sincronizzazione** dei processori nell'accesso alle risorse comuni
- Ad esempio, la coda dei processi pronti

Approccio peer



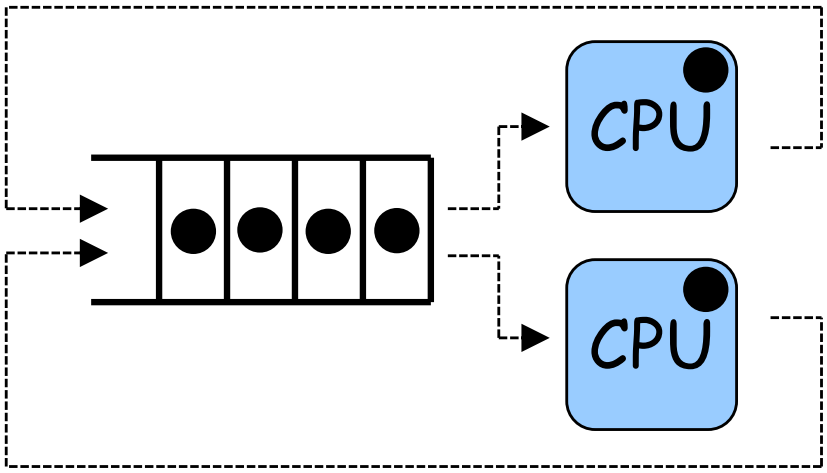
Assegnazione dei processi ai processori

- *Assegnazione statica*: ogni processo o thread è **assegnato permanentemente** ad uno dei processori
  - Ogni processore ha una **propria coda di processi pronti**
  - Approccio semplice: l'assegnazione è fatta una volta per tutte

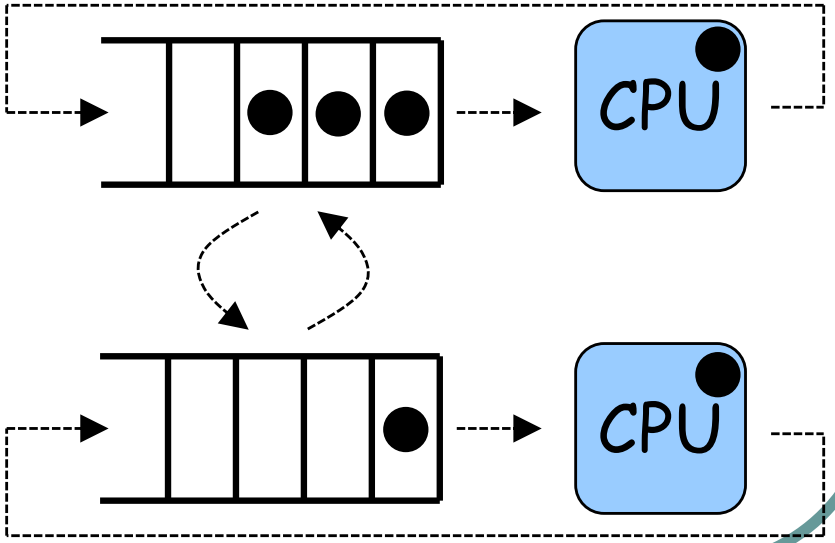
Problema: il carico potrebbe **non essere uniforme** tra i processori (dipende dal comportamento dei processi)

Assegnazione dei processi ai processori

- *Assegnazione dinamica*: durante la sua vita un processo può eseguire su processori differenti
  - **Load sharing**: Una sola coda condivisa dei processi pronti
  - **Dynamic load balancing**: Più code, una per processore, in cui i processi possono essere spostati da una coda all'altra



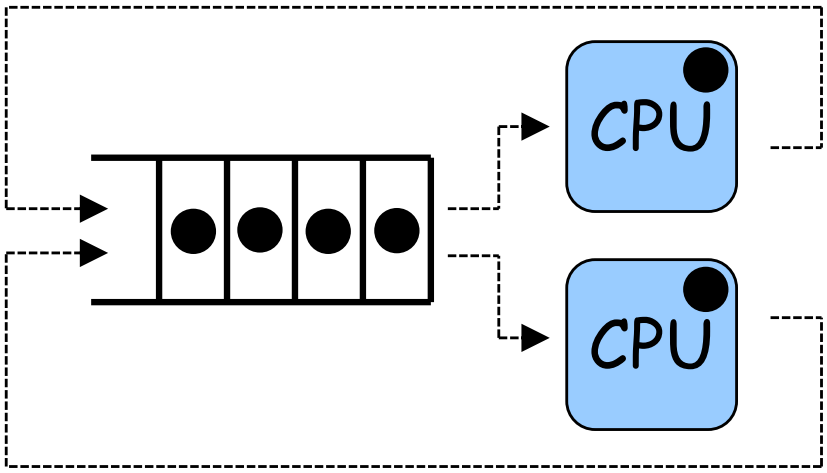
Load sharing



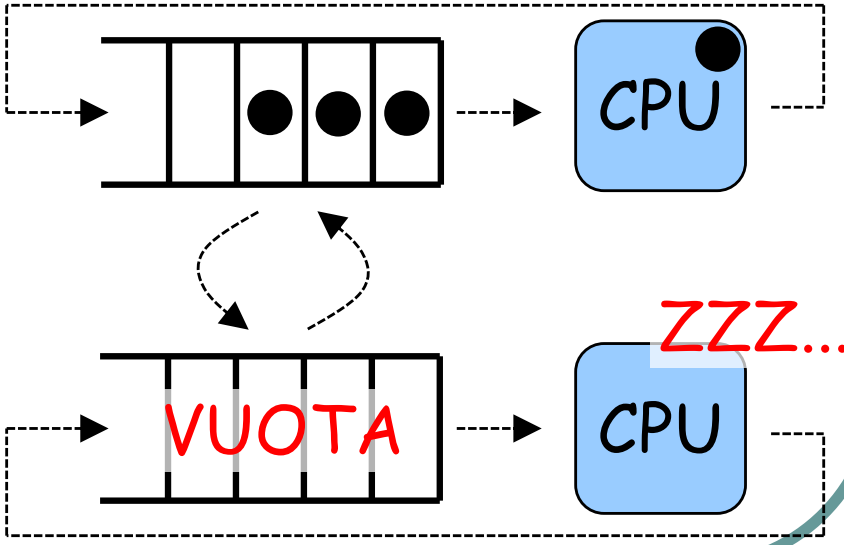
Dynamic
load balancing

Assegnazione dei processi ai processori

- Idealmente, avere una coda unica garantisce il maggior utilizzo possibile della CPU
- Con code separate, c'è un rischio (anche se piccolo) che una coda diventi vuota



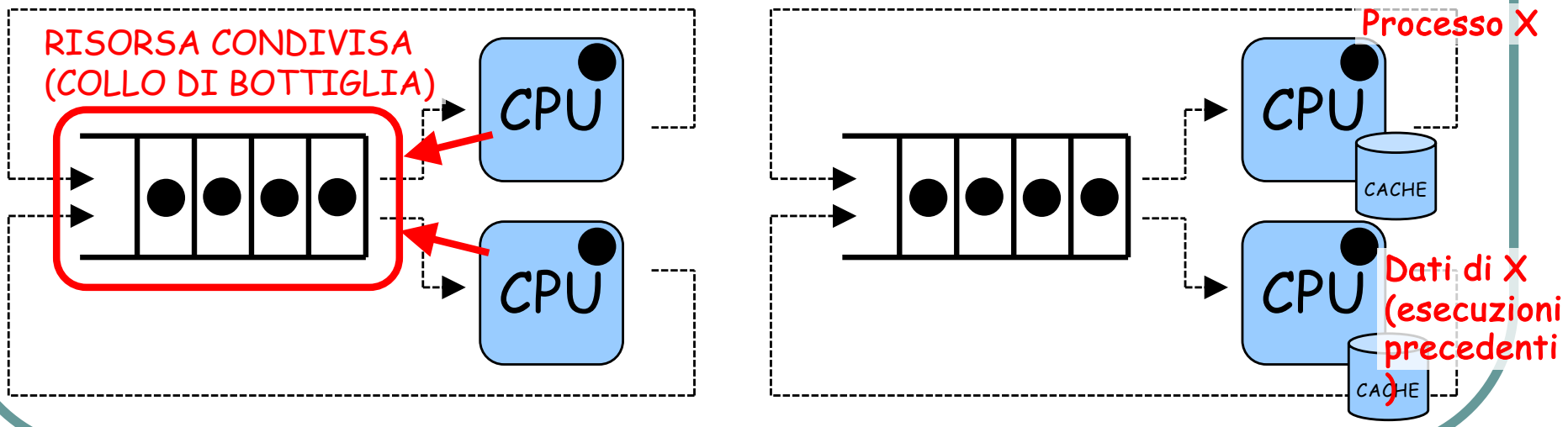
Load sharing



Dynamic load balancing

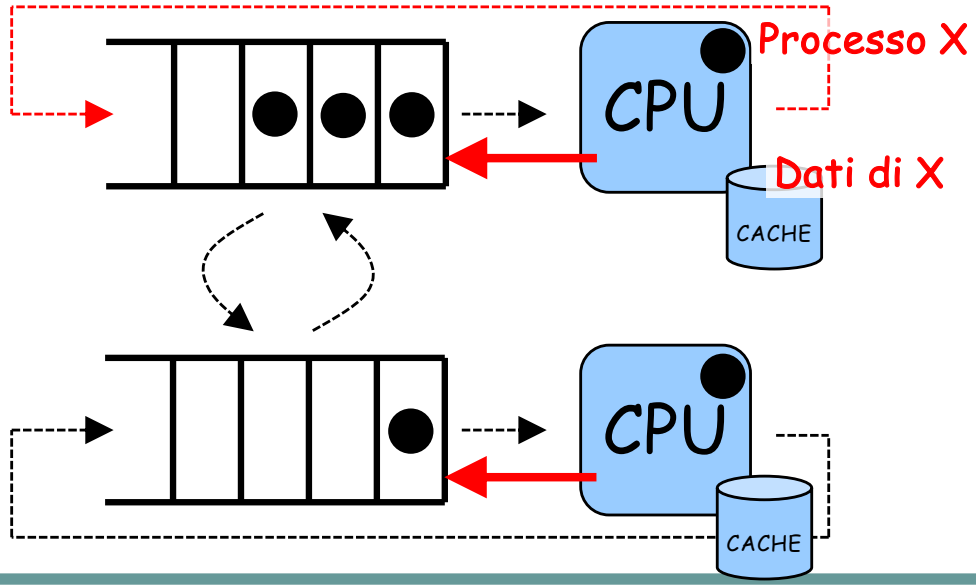
Assegnazione dei processi ai processori

- Avere una sola coda crea però due problemi:
  - La coda occupa una regione di memoria **condivisa**, che deve essere **protetta da accessi concorrenti** (mutua esclusione)
  - Non garantisce che un processo riprenda l'esecuzione sullo **stesso processore** (uso poco efficiente delle **cache dei processori**)
  - In SMP, le cache/TLB sono **locali** alle CPU/core



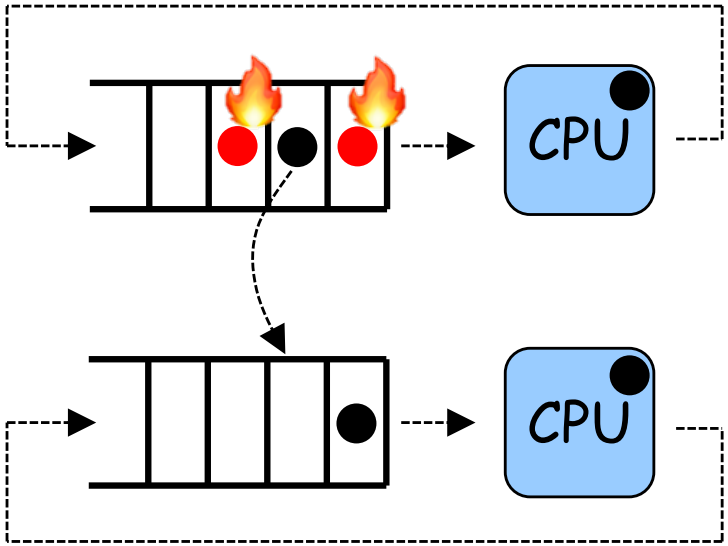
Assegnazione dei processi ai processori

- Con code multiple
  - Ogni CPU accede a una coda separata (no bottleneck)
  - I processi riutilizzano la stessa CPU (a meno di load balancing)
- **CPU affinity**: un processo tende ad eseguire più velocemente se viene riutilizzato lo stesso CPU/core

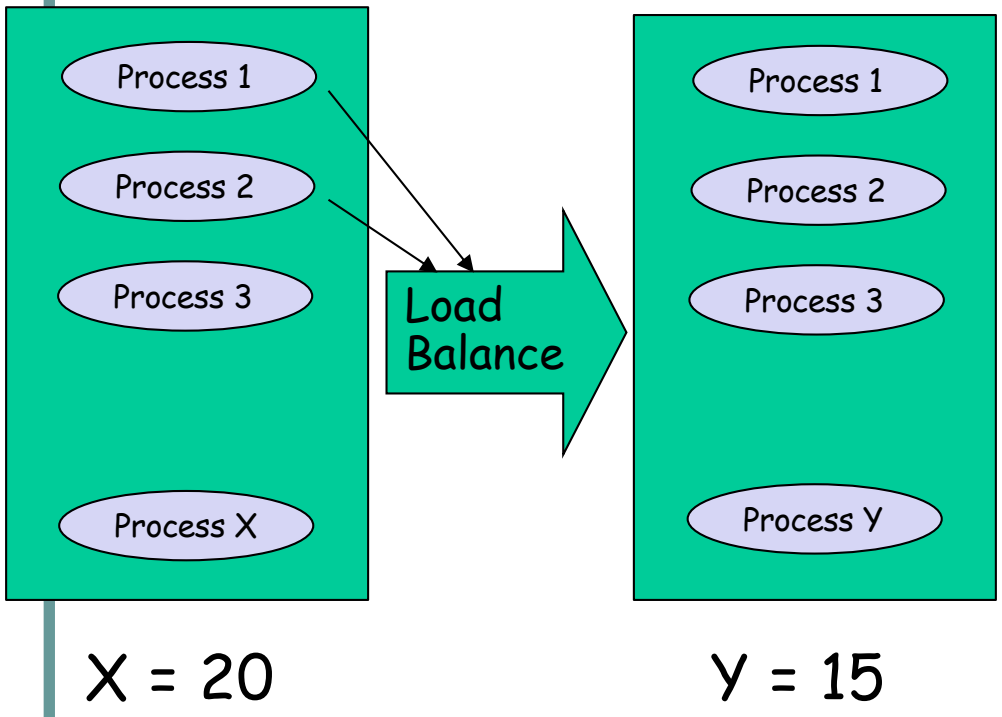


CPU affinity

- Spostare un processo/thread da un processore all'altro **danneggia il principio di località**!
- Il load balancing comporta un **impatto negativo** sui processi migrati
- È necessario **migrare i processi con minore affinità**
  - minor tempo speso in esecuzione sulla CPU
  - minor consumo di cache

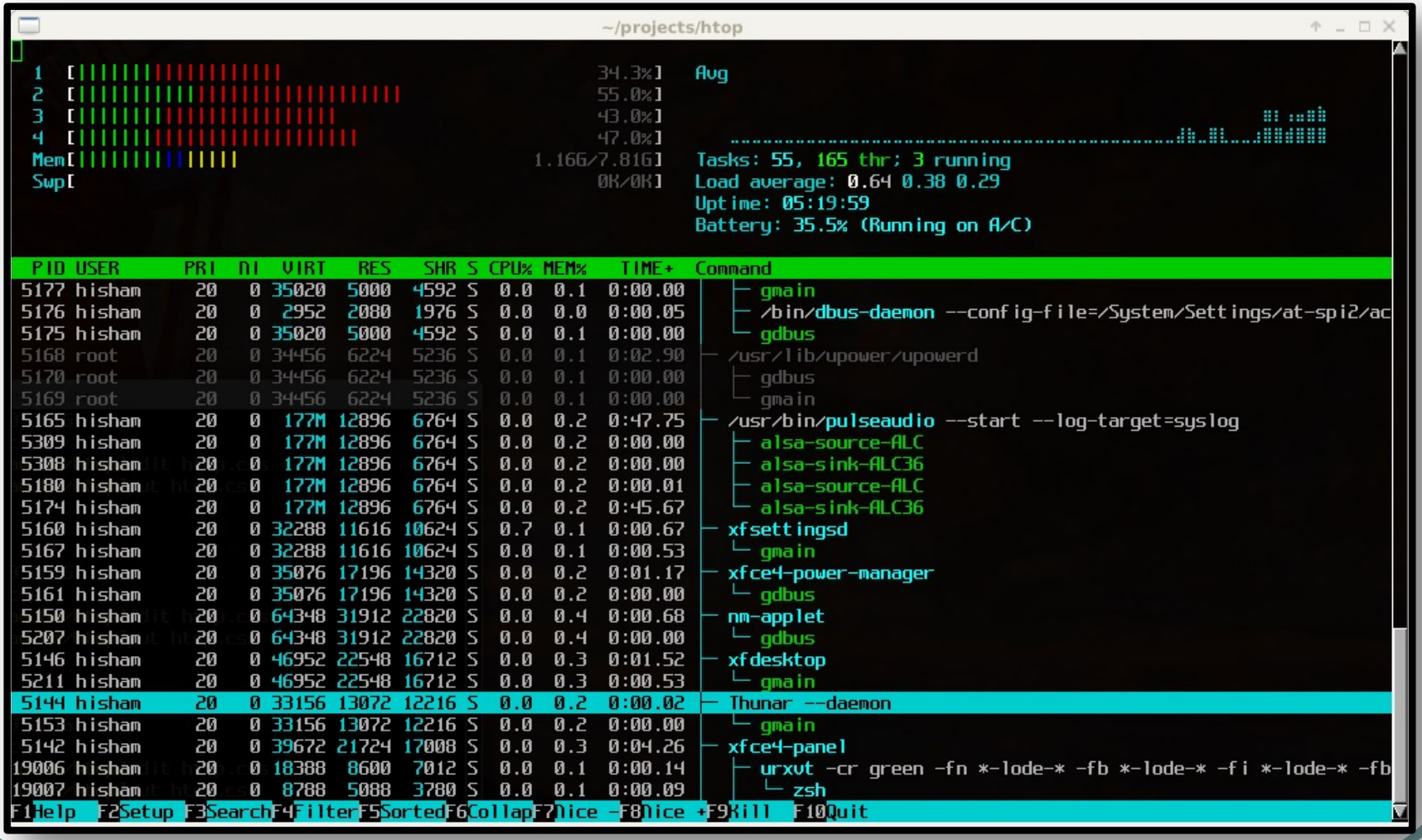


Load balancing in Linux



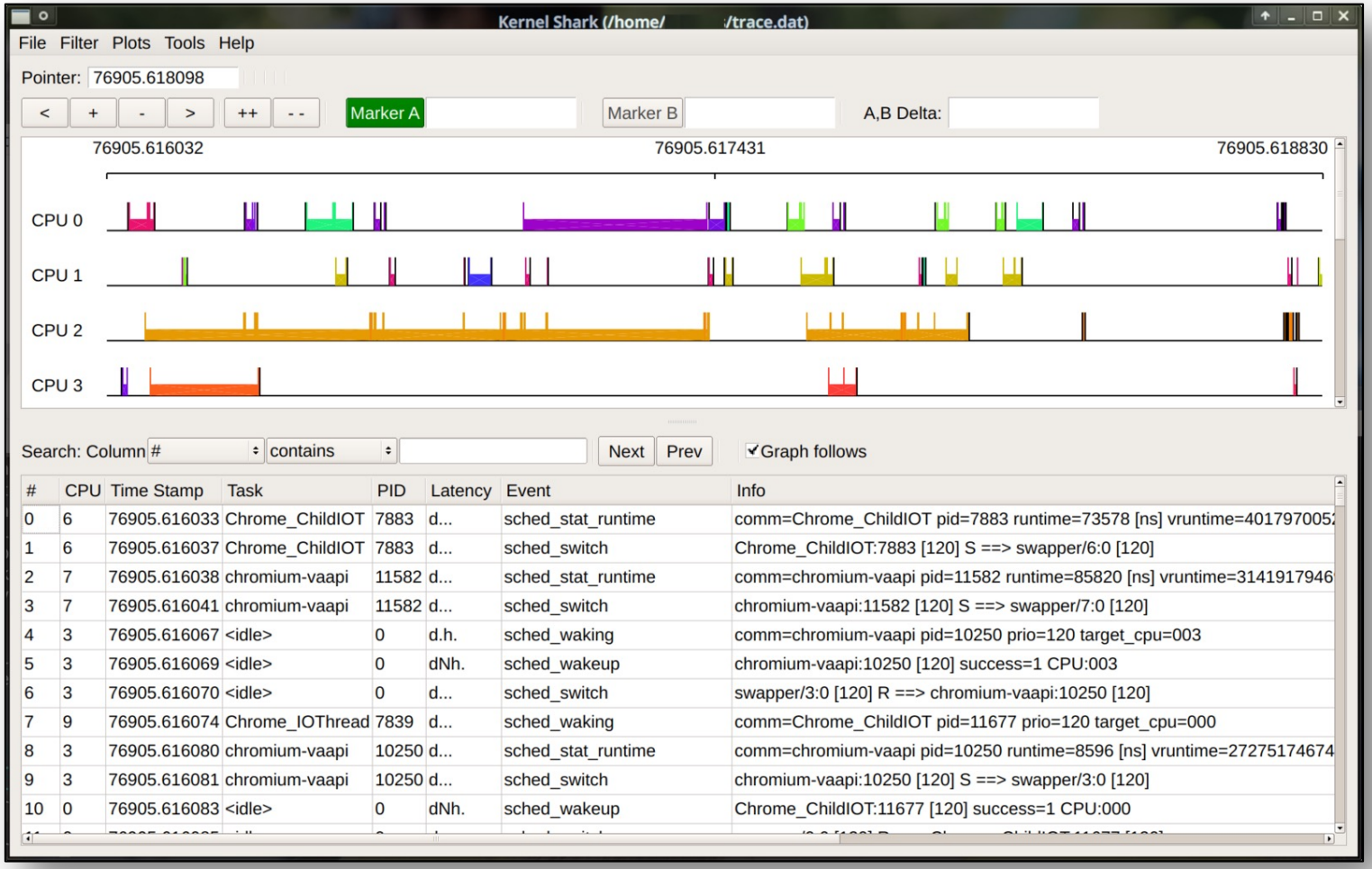
- Linux adotta il dynamic load balancing
- Il load balancing è attivato **periodicamente**, o quando una **runqueue è vuota**
- Vengono estratti dalla lista i task che non stanno eseguendo e non sono cache-hot
- L'algoritmo termina quando la runqueue con il maggior numero di task **non eccede del 25%** le altre runqueue

HTOP



https://htop.dev/

KernelShark



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	6	76905.616033	Chrome_ChildIOT	7883	d...	sched_stat_runtime	comm=Chrome_ChildIOT pid=7883 runtime=73578 [ns] vruntime=401797005
1	6	76905.616037	Chrome_ChildIOT	7883	d...	sched_switch	Chrome_ChildIOT:7883 [120] S ==> swapper/6:0 [120]
2	7	76905.616038	chromium-vaapi	11582	d...	sched_stat_runtime	comm=chromium-vaapi pid=11582 runtime=85820 [ns] vruntime=3141917946
3	7	76905.616041	chromium-vaapi	11582	d...	sched_switch	chromium-vaapi:11582 [120] S ==> swapper/7:0 [120]
4	3	76905.616067	<idle>	0	d.h.	sched_waking	comm=chromium-vaapi pid=10250 prio=120 target_cpu=003
5	3	76905.616069	<idle>	0	dNh.	sched_wakeup	chromium-vaapi:10250 [120] success=1 CPU:003
6	3	76905.616070	<idle>	0	d...	sched_switch	swapper/3:0 [120] R ==> chromium-vaapi:10250 [120]
7	9	76905.616074	Chrome_IOThread	7839	d...	sched_waking	comm=Chrome_ChildIOT pid=11677 prio=120 target_cpu=000
8	3	76905.616080	chromium-vaapi	10250	d...	sched_stat_runtime	comm=chromium-vaapi pid=10250 runtime=8596 [ns] vruntime=27275174674
9	3	76905.616081	chromium-vaapi	10250	d...	sched_switch	chromium-vaapi:10250 [120] S ==> swapper/3:0 [120]
10	0	76905.616083	<idle>	0	dNh.	sched_wakeup	Chrome_ChildIOT:11677 [120] success=1 CPU:000

https://www.kernelshark.org/

Quiz

1. Quale dei seguenti approcci favorisce la CPU affinity?

   - ◯ Load sharing (una sola coda condivisa di processi pronti)
   - ◯ Dynamic load balancing (tante code di processi pronti, una per processore)

https://forms.office.com/r/uFfrJTSvVc