NOS
Višeprocesorski sustavi

07
Svojstva višeprocesorskih sustava

Zašto višeprocesorski sustavi?

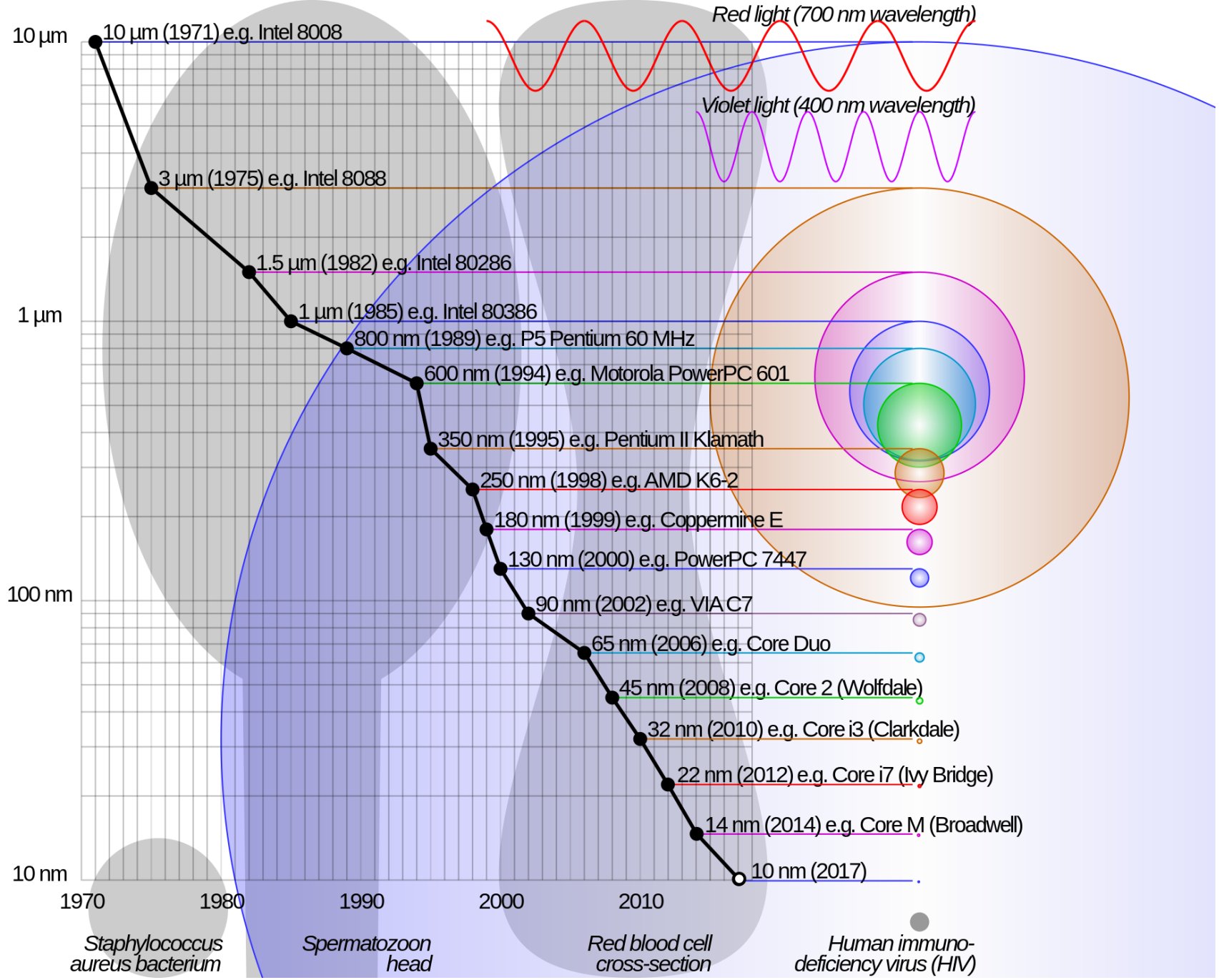
Vrste višeprocesorskih sustava

Optimizacije, područja primjene

7.1. Zašto višeprocesorski sustavi?

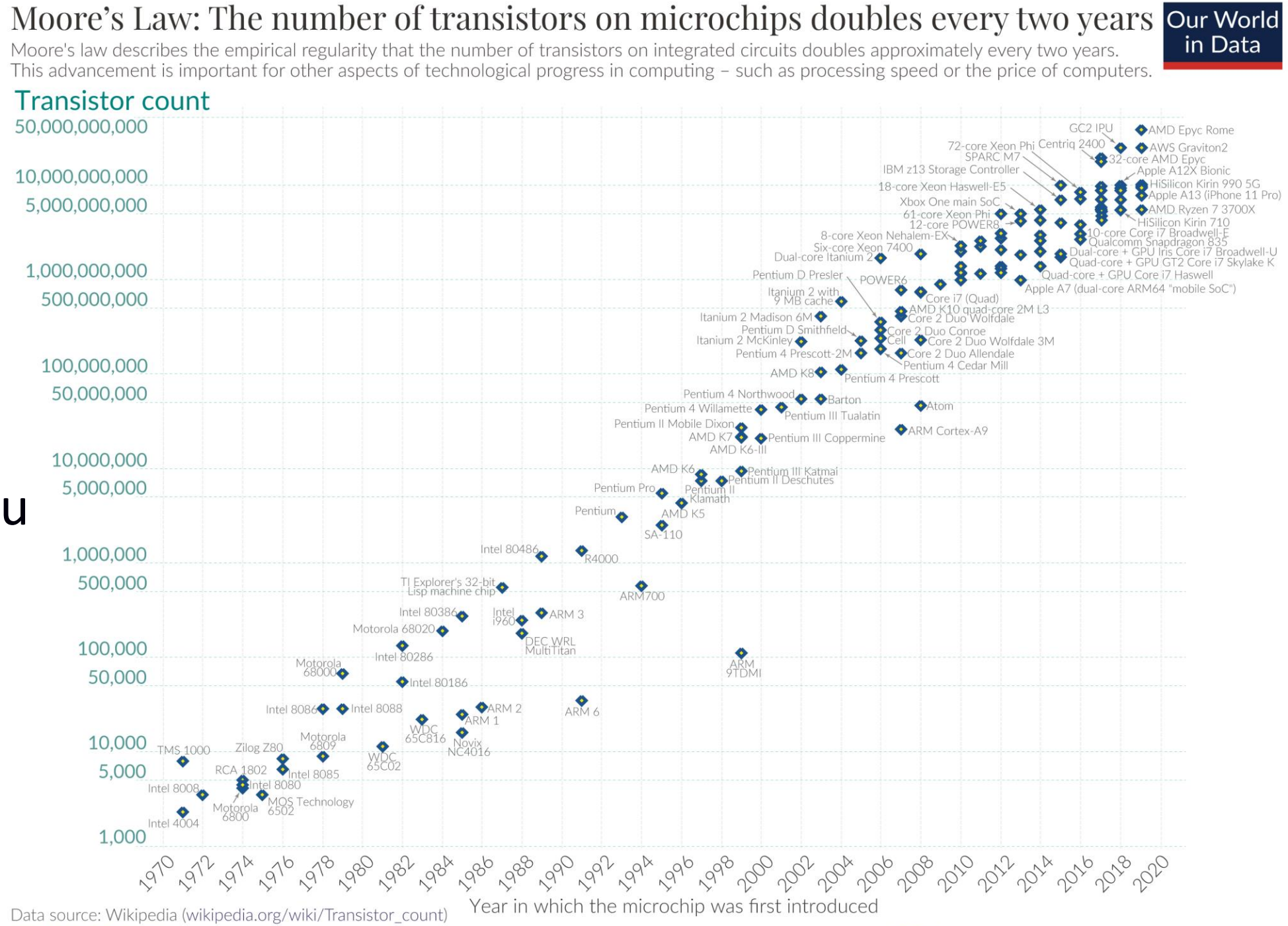
- programi moraju biti višedretveni da iskoriste višeprocesorske sustave
  - višedretveno programiranje je značajno teže
  - treba osmisliti efikasnu sinkronizaciju, komunikaciju, raspoređivanje
- razlog je poluvodička tehnologija – povećanje frekvencije nije efikasno!
  - u prošlosti (do cca 2000.) frekvencija se udvostručavala svake dvije godine
  - od tada gotovo da i nema značajna pomaka
  - ali razvojni proces i dalje napreduje, tranzistori su sve manji, više ih stane na jedan čip, manje troše energije (jedino frekvencija ne raste očekivano)
  - umjesto jednog procesora na isti čip stavlja se više „procesora”
    - sad se „procesor” (jedan čip) sastoji od više „jezgri” (core)
    - s aspekta OS-a to je „višeprocesorski sustav”

Razvoj poluvodičke tehnologije kroz godine

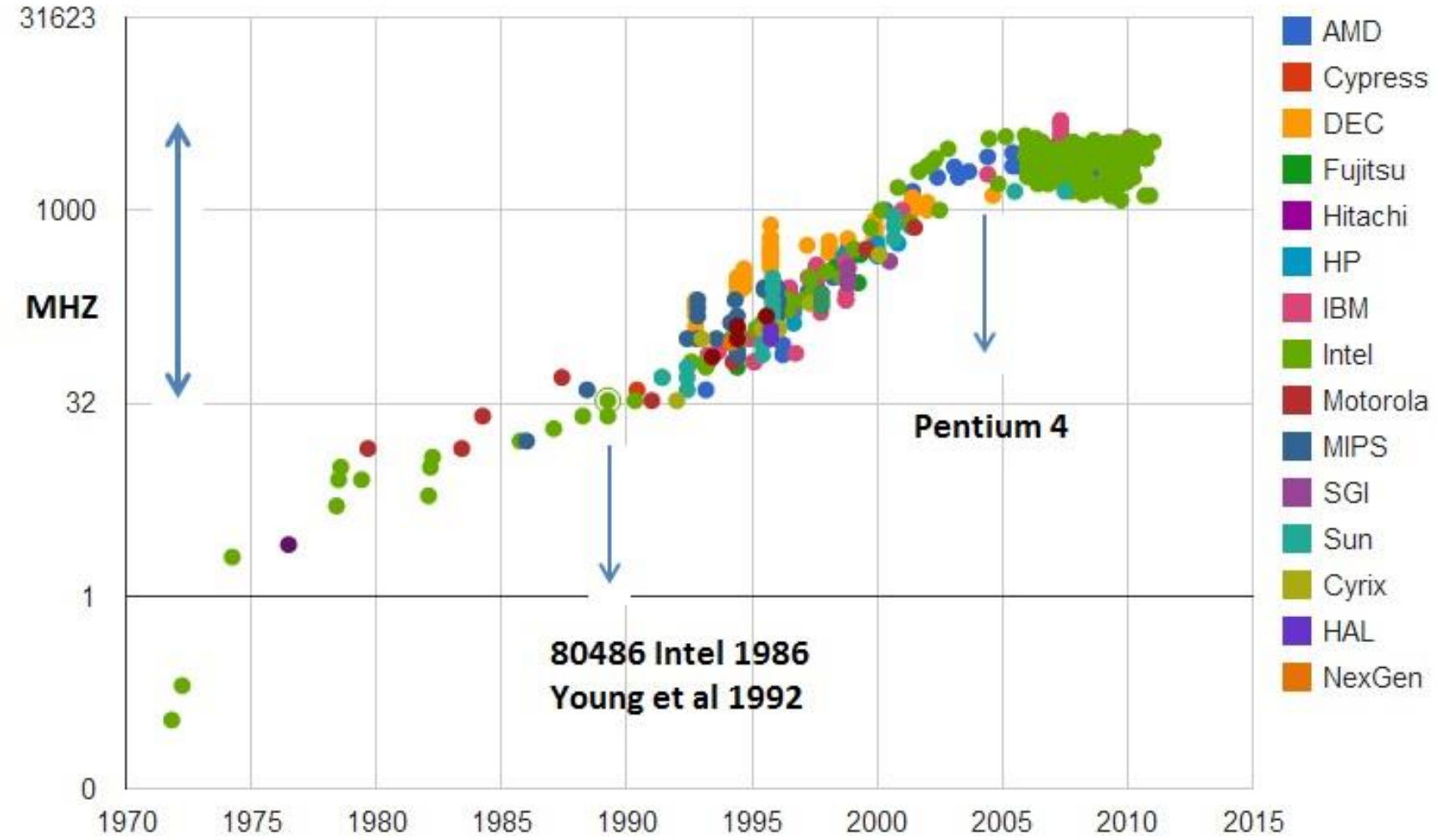


https://en.wikipedia.org/wiki/Microprocessor_chronology

Broj tranzistora na procesoru kroz godine

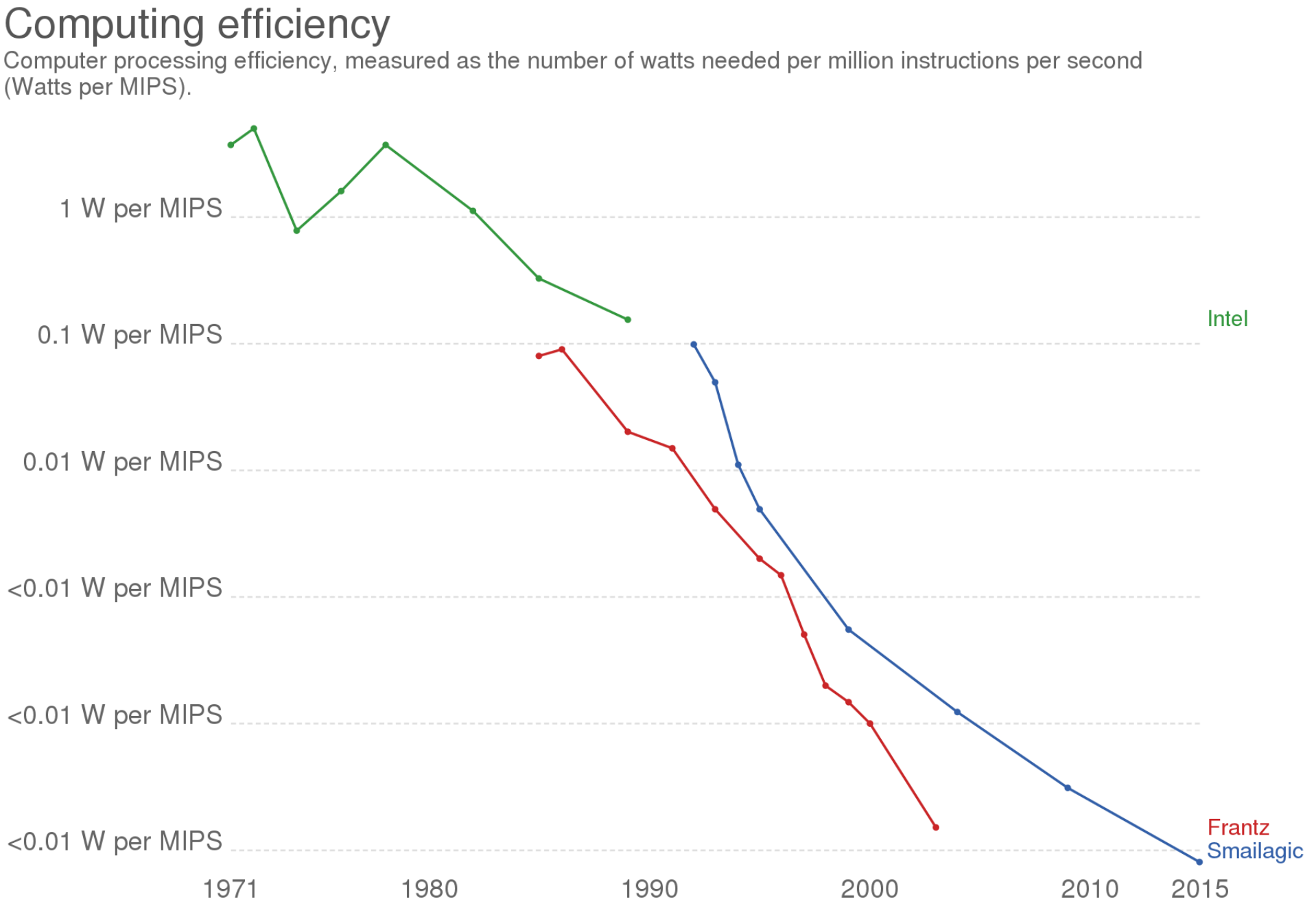


Frekvencija procesora kroz godine

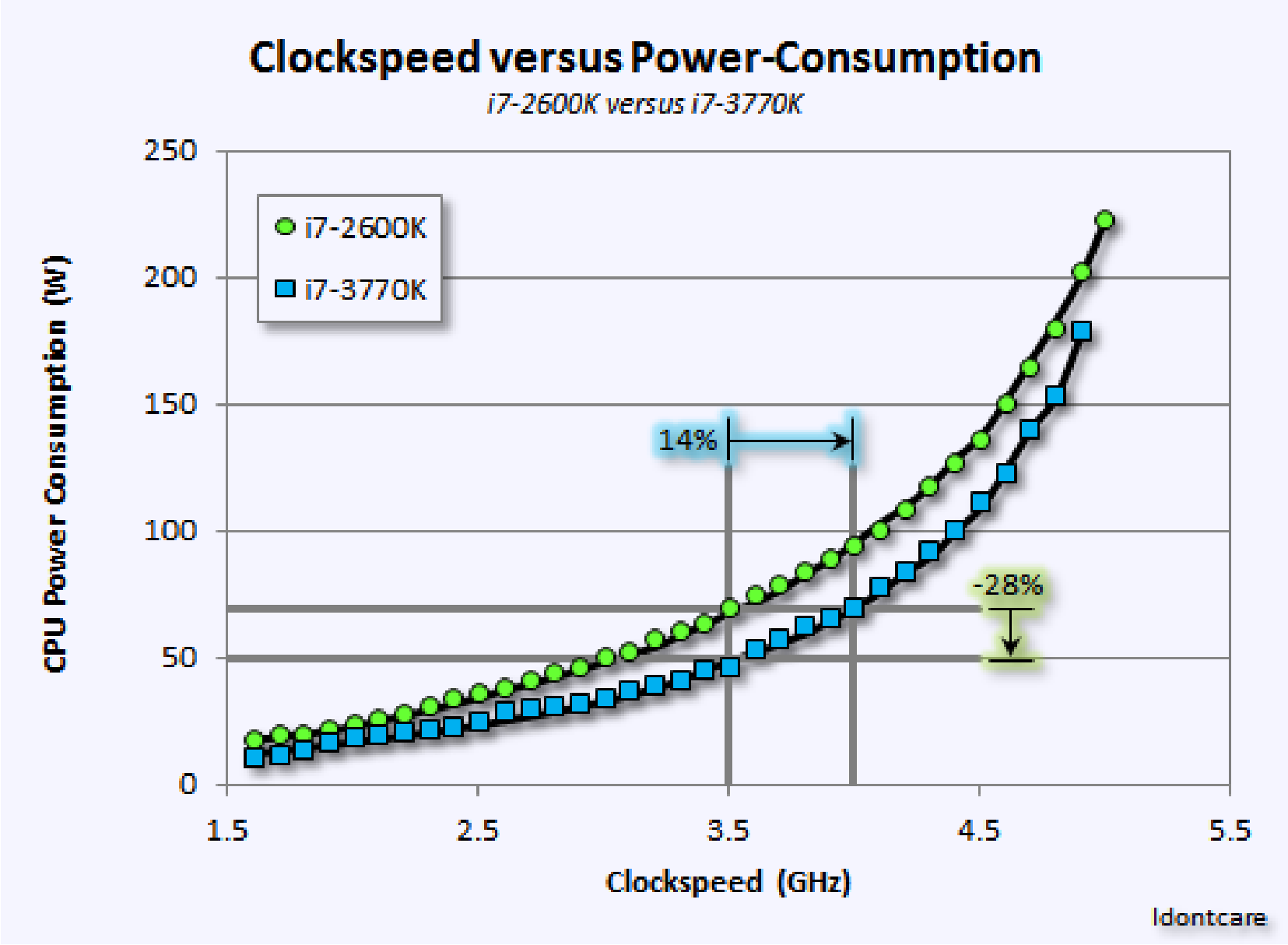


https://en.wikipedia.org/wiki/File:Clock_CPU_Scaling.jpg

Potrošnja procesora prema operacijama

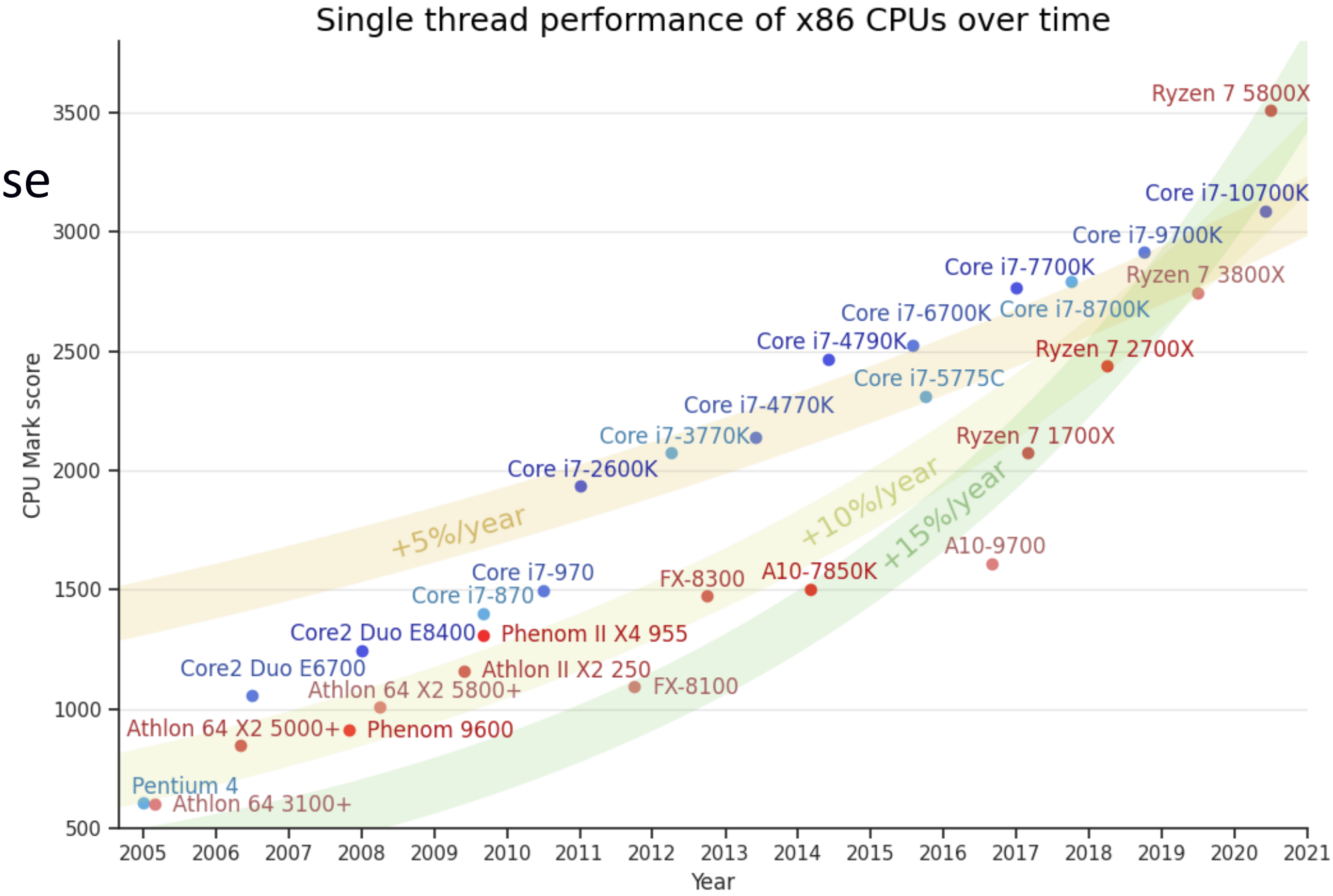


Noviji procesori troše manje na istoj frekvenciji



https://www.quora.com/Why-havent-CPU-clock-speeds-increased-in-the-last-5-years

Performanse procesora rastu i za samo jedno-dretvene aplikacije

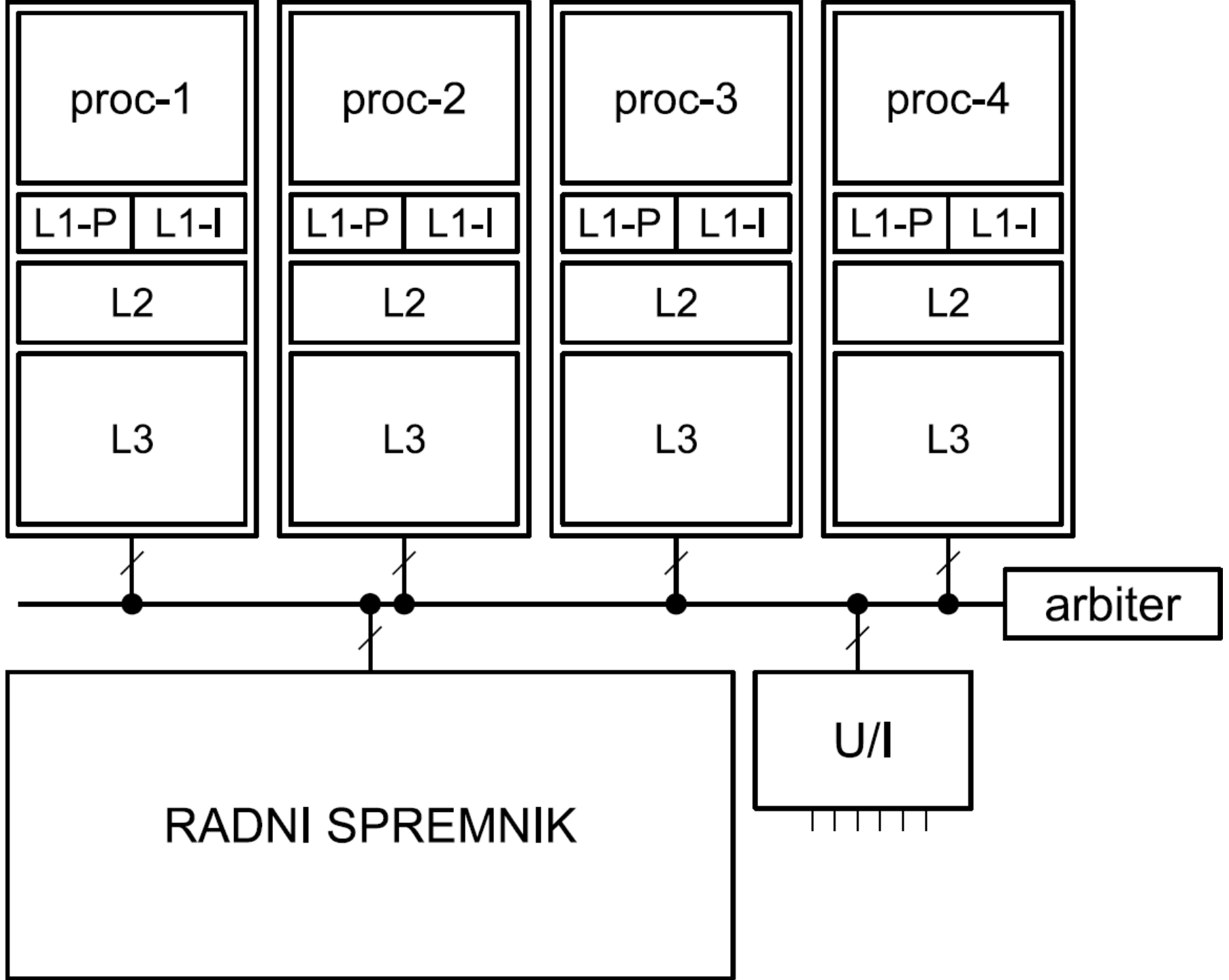


https://mlech26l.github.io/pages/2020/12/17/cpus.html

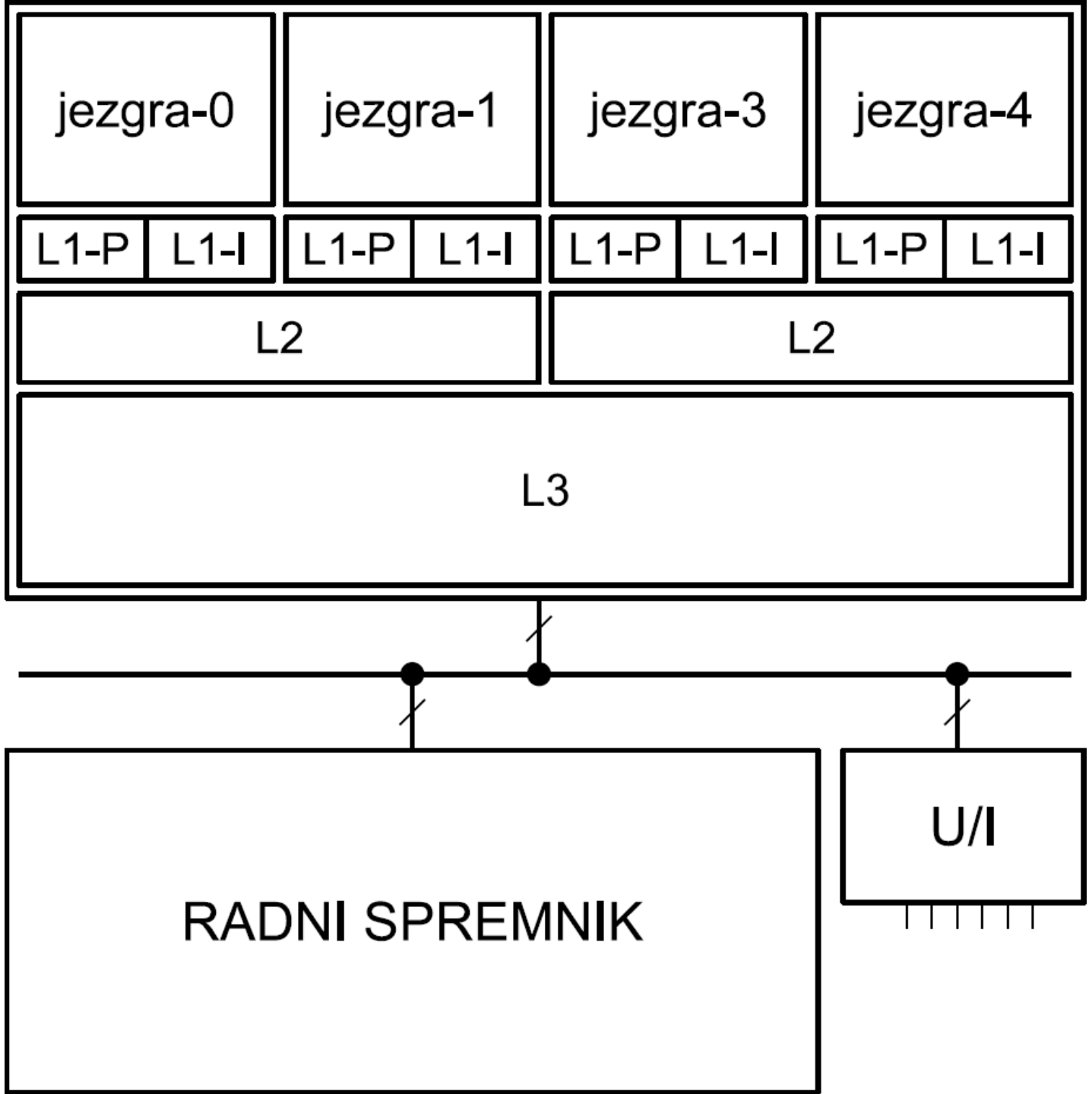
7.2. Vrste višeprocesorskih sustava

- simetrični višeprocesorski sustavi
- višejezgreni procesori
- simetrični višeprocesorski sustavi s višejezgrenim procesorima
- raspodijeljeni višeprocesorski sustavi (NUMA)

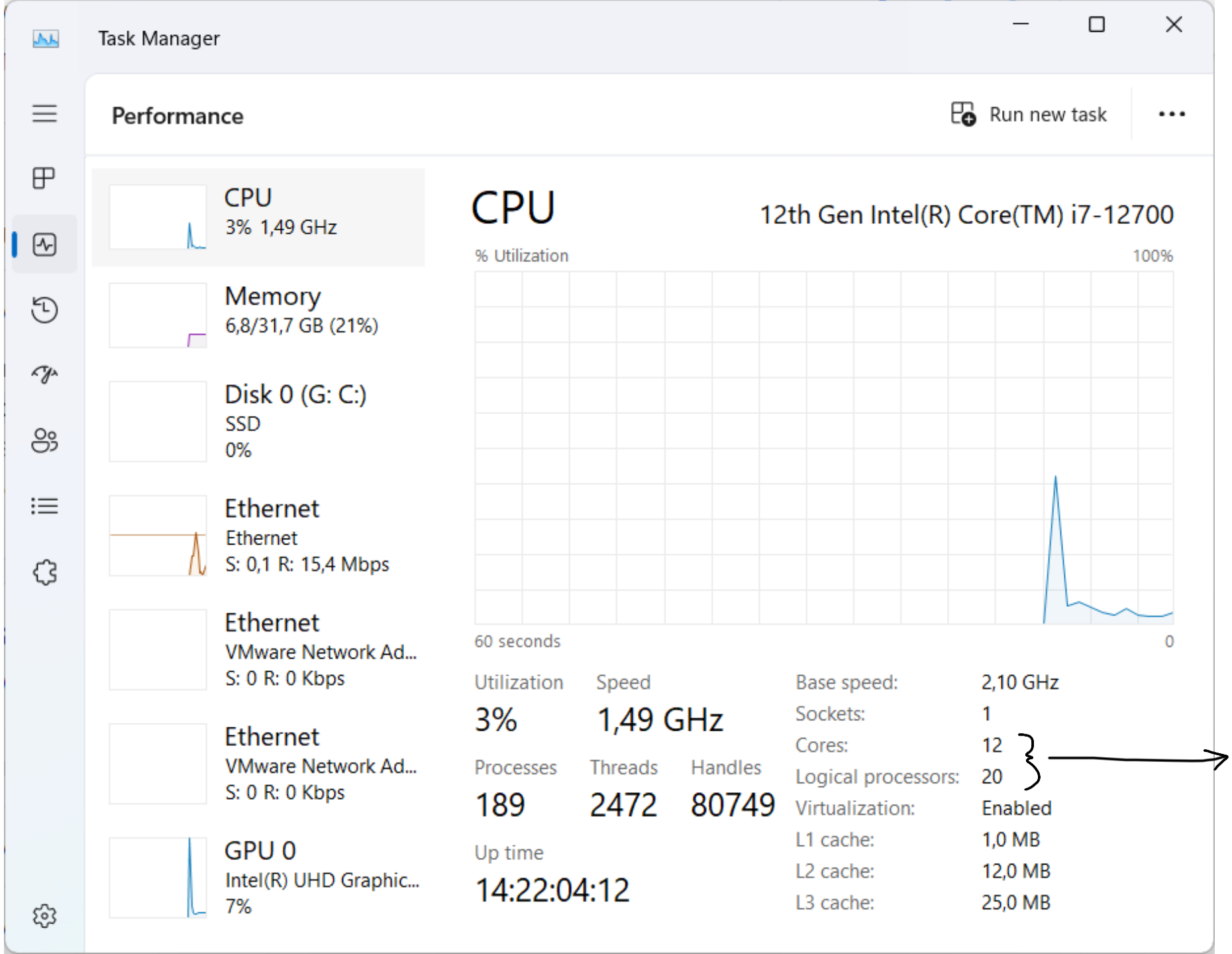
simetrični višeprocesorski sustavi (klasični v.s.)



višejezgreni procesor

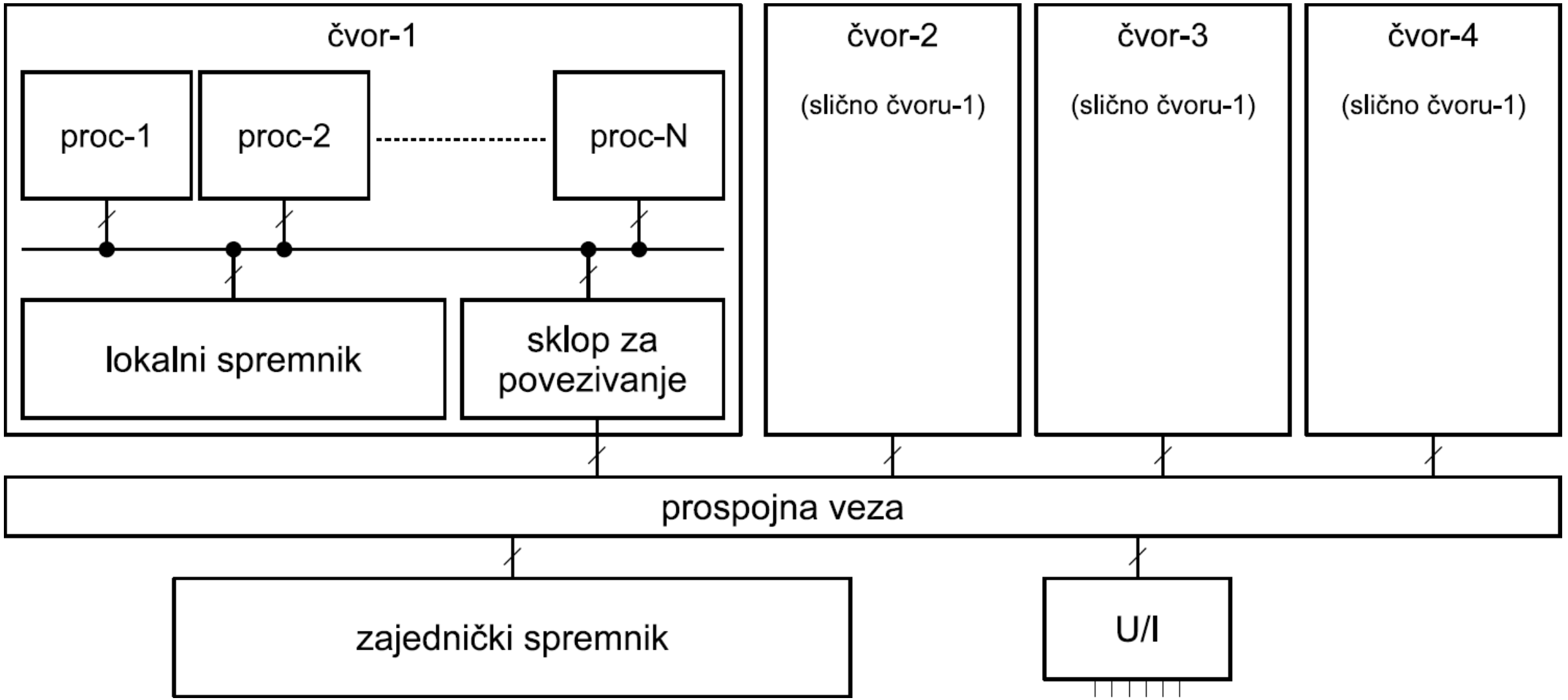


Primjer višejezg. procesora kroz Task Manager na Win.



8 + 4
P E
↓
hyperthread.

Raspodijeljeni višeprocesorski sustav (NUMA)



Simetrični/asimetrični, homogeni/nehomogeni

- simetrični/asimetrični – pristup memoriji jednako traje svima?
  - uglavnom simetrični
  - NUMA su asimetrični s raspodijeljenom memorijom
- homogeni/nehomogeni – svojstva procesora su ista?
  - uglavnom homogeni (do sada)
  - varijabilna frekvencija jezgri, ali istih svojstava
  - noviji procesori imaju različite jezgre
    - brze jezgre (P-preformance)
    - učinkovite jezgre (E-efficiency) koje troše manje struje
    - razlika nije samo u frekvenciji već i svojstvima

7.3. Što se još koristi za povećanje procesorske snage?

- Optimiranje protočne strukture
  - manje ili više elemenata u protočnoj strukturi (Pentium III → Pentium 4)
  - bolje predviđanje grananja i oportuno pripremanje i izvođenje instrukcija
  - paralelno izvođenje susjednih nepovezanih instrukcija iste dretve
- Optimiranje korištenja priručnog spremnika
  - priručni spremnik (i njegova veličina) značajno utječu na preformanse
  - procesori za poslužitelje (koji su i značajno skuplji) imaju osjetno veći priručni spremnik
  - bolji algoritmi upravljanja sadržajem povećavaju performanse
  - konzistencija podataka u višejezgrenim procesorima
    - poništavanje (engl. invalidation), prisluškivanje (engl. snooping)
- Sklopovska višedretvenost

Sklopovska višedretvenost

- engl. simultaneous multithreading, hyper-threading
- jedna jezgra ali prema van (OS-u) kao da su dvije
- procesor izvodi jednu dretvu i kada ta mora čekati kraj neke složene operacije (npr. na matematičkom koprocesoru) procesor aktivira drugu dretvu koja može koristiti druge dijelove procesora
  - obično ovakav procesor ima više dijelova koji mogu paralelno raditi
- procesor sam manipulira tim dretva (OS misli da se obje izvode)
- dobitak na performansama i do 30%
- ponekad je tu funkcionalnost potrebno isključiti:
  - takve dretve ne napreduju očekivanim tempom (ne dobiju svoj procesor)
  - moguća degradacija performansi zbog prevelikih zahtjeva prema priručnom spremniku

7.4. Okruženja primjene višeprocesorskih sustava

- Podjela okruženja prema svojstvima
  - poslužitelji
  - osobna računala (stolno računalo, prijenosnik)
  - pametni telefoni, tableti
  - ugrađena računala

- različita okruženja traže različite optimizacije (procesora, OS-a)
  - performanse (za jedan posao ili pak za više paralelnih poslova)
  - smanjena potrošnja
  - determinističko ponašanje

Problemi pri ostvarenju operacijskog sustava za višeprocesorske sustave

Paralelni rad u jezgri

Konzistentnost podataka kroz zaključavanja, atomarne operacije

Strukture podataka jezgre (za raspoređivanje)

Problemi ostvarenja OS-a u višeprocesorskom okruženju

- poželjna svojstva jezgre (problem jest kako ih ostvariti)
  - paralelni rad u jezgri (različite dretve na različitim procesorima)
  - očuvanje konzistentnosti podataka jezgre i kroz paralelni rad
  - optimalno koristiti priručne spremnike – oni značajno utječu na performanse

  - osigurati pravednu podjelu procesorskog vremena
  - omogućiti rezervaciju resursa (u sustavima gdje to treba)
  - ostvariti deterministično ponašanje (u sustavima gdje to treba)

Problemi ostvarenja OS-a u višeprocesorskom okruženju (2)

- Dopustiti dretvama da se izvode na bilo kojem procesoru ili ne?
  - uglavnom „da”, osim u posebnim situacijama
    - kad se želi rezervirati procesor za nešto drugo
    - kad se želi „uspavati neki procesor” radi uštede energije
  - postaviti afinitet dretvama – na kojim procesorima da se izvode
    - dretve mogu same tražiti postavljanje nekog afiniteta, a može se to i izvana
- Upravljanje napravama
  - koristiti dedicirani procesor ili bilo koji je slobodan/prikladan u datom trenu?
    - „bilo koji” je OK, omogućuje najmanje degradacije performansi dretvama
    - dedicirani daje bolji odziv na vanjske događaje

8.1. Očuvanje konzistentnosti podataka jezgre

- Osnovni načini:
  - zaključavanja (npr. mutex, semafori, spinlock)
  - atomarne operacije (npr. `atomic_add_fetch`)
  - „oprezno” korištenje (posebne instrukcije procesora, upute prevoditelju)

Zaključavanja u jezgri

- mehanizam prekida osigurava međusobno isključivanje na jednom procesoru – nije dostatno u višeprocesorskim sustavima
- pokušati koristiti što je moguće manje zaključavanja da se i jezgrine funkcije mogu paralelno izvoditi na različitim procesorima i dretvama
  - zaključavaju se puno manje strukture podataka (ne „big kernel lock”)
  - svaka takva struktura ima svoj „ključ”
- zaključavanje može biti:
  - blokirajuće – dretva se miče s procesora (mutex, semafor, red, …)
  - neblokirajuće – radnim čekanjem – spinlock
- odabir mehanizma (blokirajuće ili neblokirajuće) ovisi o mnogo stvari
  - gdje se koristi ta struktura (u atomarnom kodu ili samo u kodu s kontekstom)
  - trajanje kritičnog odsječka – kratko/duže i sl.

Blokirajuće i neblokirajuće - svojstva

- Ako se očekuje dulje trajanje zaključavanja onda (ako je moguće) koristiti blokirajuće zaključavanje (npr. `mutex_lock`)
  - radno čekanje bi bilo neefikasno
- U protivnom bolje odabrati radno čekanje
  - problem blokirajućeg je što ima puno kućanskih poslova
    - spremi kontekst, odaberi drugu dretvu, obnovi kontekst
    - to ponekad povlači i dodatne poslove s priručnim spremnicima i straničenjem
  - radno čekanje isto ima probleme (osim neefikasnog trošenja procesora)
    - loše utječe na sabirnicu – cijelo vrijeme se traži ažuriranje podataka
- Za vrlo kratke operacije bolje koristiti atomarne operacije/instrukcije

Atomarne operacije

- Primjeri (iz predmeta OS) koji koriste dva uzastopna sabirnička ciklusa
  - TAS – test-and-set (ispitaj i postavi)
  - SWP - zamijeni
  - CAS – compare-and-swap
- Ovakvim i sličnim instrukcijama ostvariti neke jednostavnije operacije
  - `atomic_load, atomic_store, atomic_exchange, atomic_compare_exchange, atomic_add_fetch, atomic_fetch_add, atomic_test_and_set`

Atomarne operacije (2)

- samo izvođenje ovakvih operacija nad operandima nije dovoljno
  - mora se osigurati i redoslijed prethodnih/slijedećih operacija
  - procesori rade *out-of-order* optimizacije što ovdje predstavlja problem
  - isto može napraviti i prevoditelj!
- prevoditelju se posebnim naredbama treba reći da poštuje redoslijed
  - on to ostvaruje ubacivanjem dodatnih instrukcija u program
  - te instrukcije „koštaju” – dodatno troše vrijeme
    - manje efikasno se koristi protočna struktura
    - manje efikasno se koristi priručni spremnik
    - više instrukcija
    - više zaključavanja sabirnice od strane jednog procesora

Primjer problema u paralelnom radu

```
x = 0
y = 0
Dretva 1 {          Dretva 2 {
  x = 1               ispiši(y)
  y = 2               ispiši(x)
}                   }
```

- uz pretpostavku da Dretva 2 ipak najprije ispiše y a potom x
  - sve kombinacije ispisa su ipak moguće!!!
  - 0 0, 2 1, 0 1, ali i **2 0** (kad D1 prvo obavi y = 2)

Vrste konzistencija

- Konzistencija nad slijedom operacija (total ordering)
  - najstroža, najzahtjevnija, ali i „najsigurnija”
  - (pita se od studenata!)
- Konzistencija nad jednom operacijom (relaksirana) (info)
  - gleda se samo označena varijabla
- Konzistencija nad povezanim varijablama (acquire/release) (info)
  - pohrana (release) nakon svih prethodnih, dohvat prije idućih (označenih i ostalih)
- Konzistencija nad nepovezanim varijablama (consume/release) (info)
  - pohrana (release) nakon svih prethodnih, ali samo povezanih s označenim
  - slično i s dohvatom

Konzistencija nad slijedom operacija - total ordering

1. poštuje se redoslijed posebno označenih operacija (sa `var.atomarno_nešto*`)
2. operacije (neoznačene) koje su navedene prije označene moraju biti dovršene prije nego li označena počne
3. operacije (neoznačene) koje su navedene poslije označene ne smiju započeti prije nego li označena završi

Primjer:

```
a = x = y = 0
Dretva 1 {
   a = 5
   x.atomarno_spremi(1)
   y.atomarno_spremi(2)
}
```

```
Dretva 2 {
   ispiši(y.atomarno_dohvati())
   ispiši(x.atomarno_dohvati())
   ispiši(a)
}
```

- mogući ispisi: 0 0 0, 0 0 5, 0 1 5, 2 1 5

Konzistencija nad jednom operacijom (relaksirana) (info)

- održava konzistentnost samo nad označenim podacima (pojedinačno)
  - ako jedna dretva napravi *spremi* prije nego druga *pročitaj* onda se mora pročitati spremljena vrijednost
  - to inače ne mora biti tako zbog privremene pohrane u registru ili priručnom spremniku

```
a = x = y = 0
Dretva 1 {
   a = 5
   x.atomarno_spremi(1, RELAXED)
   y.atomarno_spremi(2, RELAXED)
}
```

```
Dretva 2 {
   ispiši(y.atomarno_dohvati(RELAXED))
   ispiši(x.atomarno_dohvati(RELAXED))
   ispiši(a)
}
```

- x i y su nezavisni, ne čuva se redoslijed njihova korištenja
- mogući ispisi: 0 0 0, 0 0 5, 0 1 5, 2 1 5, ali i: 0 1 0, 2 0 0, 2 1 0, 2 0 5

Konzistencija nad jednom operacijom (relaksirana) (2) (info)

```
x = 0

Dretva 1 {
   a = 1
   x.atomarno_spremi(a, RELAXED)
   b = 2
   x.atomarno_spremi(b, RELAXED)
}
```

```
Dretva 2 {
   ispiši(x.atomarno_dohvati(RELAXED))
   ispiši(x.atomarno_dohvati(RELAXED))
}
```

- ovdje je očuvan redoslijed jer se koristi ista varijabla
- mogući ispisi: 0 0, 0 1, 1 1, 1 2, 2 2

Konzistencija nad povezanim varijablama (acquire/release) (info)

- pod “povezane varijable” misli se na korištenje globalnih varijabli koje nisu u dohvati/pohrani već neposredno prije/poslije
- pri spremanju koristiti ***release***
  - osigurava da se ova operacija obavi **nakon** svih prethodnih operacija **pohrane** u memoriju (da procesor ne bi izmijenio redoslijed i neku instrukciju za pohranu koja je u kodu prije napravio nakon ove operacije)
- pri dohvatu koristiti ***acquire***
  - osigurava da se ova operacija obavi **prije** svih slijedećih operacija **čitanja**

- na neki način stvara se zavisnost (uređenost) operacija među dretvama koje koriste ovakve operacije

Konzistencija nad povezanim varijablama (acq./rel.) (2) (info)

```
a = 0
b = 0
x = 0
y = 0
```

```
Dretva 1 {
   a = 5
   x.atomarno_spremi(1, RELEASE)
   y.atomarno_spremi(2, RELEASE)
   b = 7
}
```

```
Dretva 2 {
   ispiši(y.atomarno_dohvati(ACQUIRE))
   ispiši(x.atomarno_dohvati(ACQUIRE))
   ispiši(a)
   ispiši(b)
}
```

- „a” pa „x” pa „y”
- „b” nije uređen, može prije/poslije (kod Dretve 1)

Konzistencija nad nepovezanim varijablama (consume/release) (info)

- Slično kao i acquire/release, ali se gledaju samo operacije koje su povezane s ovom - koriste iste podatke/memoriju
- za razliku od relaksirane, ovdje se poštuje redoslijed operacija dohvati/pohrani (samo) označenih operacija

```
a = b = x = y = 0
z = NULL
Dretva 1 {
   a = 5
   b = 7
   x.atomarno_spremi(1, RELEASE)
   y.atomarno_spremi(2, RELEASE)
   z.atomarno_spremi(&b, RELEASE)
}
```

```
Dretva 2 {
  ispiši(y.atomarno_dohvati(CONSUME))
  ispiši(x.atomarno_dohvati(CONSUME))
  w = z.atomarno_dohvati(CONSUME)
  ispiši(*w)
  ispiši(a)
}
```

Atomarnost pohrane i čitanja

- prethodne operacije „koštaju” (dodatni overhead)
- zato se izbjegavaju kad nisu neophodne
- ponekad je problem i samo čitanje/spremanje jednog podatka
  - priručni spremnici, njihova sinkronizacija i slično
  - može biti razlomljeno na više sabirničkih ciklusa (možda ne i susjednih!)
  - u različitim arhitekturama postoje različita rješenja kako ipak osigurati ispravnost i nedjeljivost tih osnovnih operacija
    - u kodu jezgre Linuxa koriste se makroi: `READ_ONCE, WRITE_ONCE`
    - u C-u varijable se označavaju s `volatile`
- (info) Read-copy-update (RCU) – kada se podaci uglavnom čitaju a rjeđe modificiraju
  - umjesto zaključavanja (npr. reader/writer lock) koristiti dodatne strukture

8.2. Strukture podataka jezgre: raspoređivanje dretvi

- više aktivnih dretvi
- red pripravnih = za svaki procesor po jedan „red”
  - bolje iskorištenje priručnog spremnika (*hot cache*)
    - dretve koje „odu” i „brzo” se vrate na isti procesor tamo možda još nađu svoje podatke – ne treba ih ponovno dohvaćati iz memorije
  - sitnija zrnatost pri zaključavanju pri raspoređivanju (*zaključaj samo svoj red*)
  - potrebne su dodatne operacije zbog više redova pripravnih
    - balansiranje – da sustav bude pravedan prema svim dretvama
    - *gurni/povuci* kad se koristi prioritetni raspoređivač (o tome više kasnije)
  - uzeti u obzir hijerarhiju jezgri/procesora; koje jezgre dijele koje dijelove priručnog spremnika i slično – mogu se dobiti osjetno bolje performanse sustava u nekim situacijama

Ostale strukture podataka jezgre

- uglavnom zajedničke za cijeli sustav
- svaka „cjelina” ima svoj ključ za minimalno zaključavanje

- podaci se „skrivaju”, koriste kroz sučelja modula
  - eventualne promjene u strukturama su onda vidljive samo iznutra
- izuzeci su neke često korištene varijable/kazaljke, npr.
  - `jiffies` – sat u broju otkucaja
  - `current` – kazaljka na opisnik aktivne dretve (u jezgri)

Raspoređivanje dretvi u višeprocesorskim sustavima

Pretpostavka: jedan sustav jedan OS

- moglo bi i drukčije: jedan procesor jedan OS
  - nepraktično, neefikasno
  - nešto slično je virtualizacija
  - neće se razmatrati u nastavku

9.1. Raspoređivači

- iste ideje se koriste za jednoprocesorske i višeprocesorske sustave
- raspoređivači se dijele u dvije klase
  - raspoređivanje vremenski kritičnih poslova (real-time)
  - raspoređivanje nekritičnih (običnih) poslova

- svojstva kritičnih poslova
  - objašnjena se drugdje npr. u predmetu *Sustavi za rad u stvarnom vremenu*
  - ukratko: treba nešto napraviti unutar zadanih vremenskih ograničenja; inače posljedice mogu biti ozbiljne (ekonomske, stradanja ljudi i sl.)
  - najčešće se ovaj problem rješava raspoređivanjem prema prioritetu

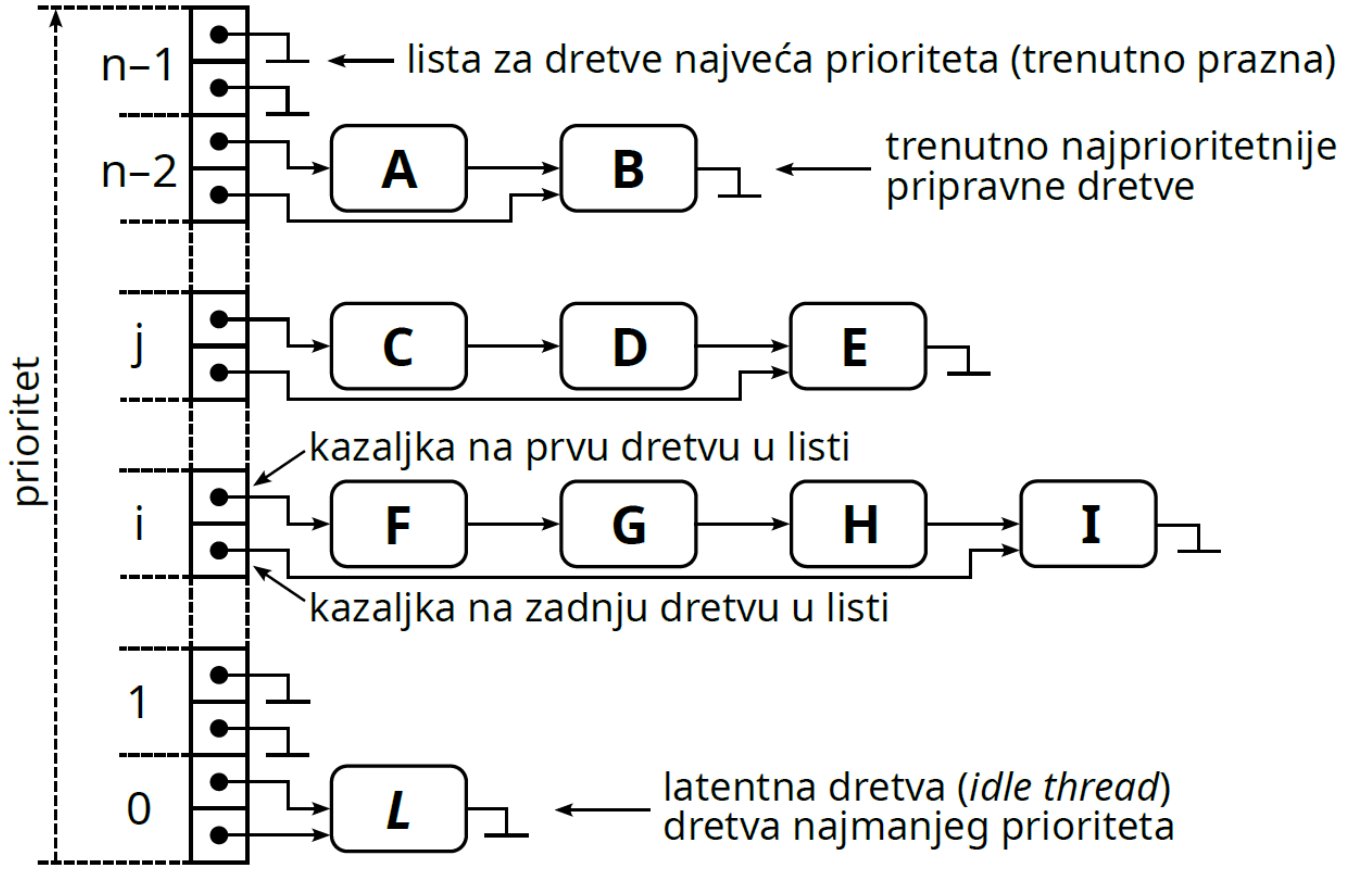
Raspoređivači za vremenski kritične poslove

- Uobičajeni raspoređivači implementirani u OS-eveima
  - SCHED_FIFO – prioritet pa red prispijeća
  - SCHED_RR – prioritet pa podjela vremena
  - SCHED_DEADLINE – prema trenucima kad moraju biti gotovi (Linux samo)
  - (info) SCHED_SPORADIC – prema prioritetu, ali uz rezervacije vremena
- Prioritet se pridjeljuje prema važnosti posla ili na druge načine
  - npr. za skup periodičkih poslova prema mjeri ponavljanja (RMPA)
    - poslovi koji se češće javljaju dobivaju veći prioritet
- Kritične dretve (real-time) dobivaju procesorsko vrijeme prije nekritičnih
  - greška u njima može „srušiti sustav”
  - stoga ih može pokrenuti samo povlašteni korisnik (root/admin)

Raspoređivanje dretvi prema prioritetu

- raspoređuju se pripravne dretve
- pripravna dretva najveće prioriteta se uvijek odabire za izvođenje
  - tj. N najprioritetnijih na N-procesorskom sustavu
- čim se takva dretva pojavi ona istisne manje prioritetnu dretvu s procesora
- Raspoređivači (većinom) ostvareni u (UNIX) OS-evima
  - SCHED_FIFO – prema prioritetu pa po redu prispijeća
  - SCHED_RR – prema prioritetu pa kružnom podjelom vremena
  - **ime dobivaju po sekundarnom kriteriju** (jer prvi je prioritet)
- Uz prikladnu strukturu podataka (red pripravnih) operacije raspoređivača su u složenosti O(1) !

Primjer reda pripravnih pri raspoređivanju prema prioritetu



Slika 5.1. Primjer strukture podataka jezgre za raspoređivanje prema prioritetu

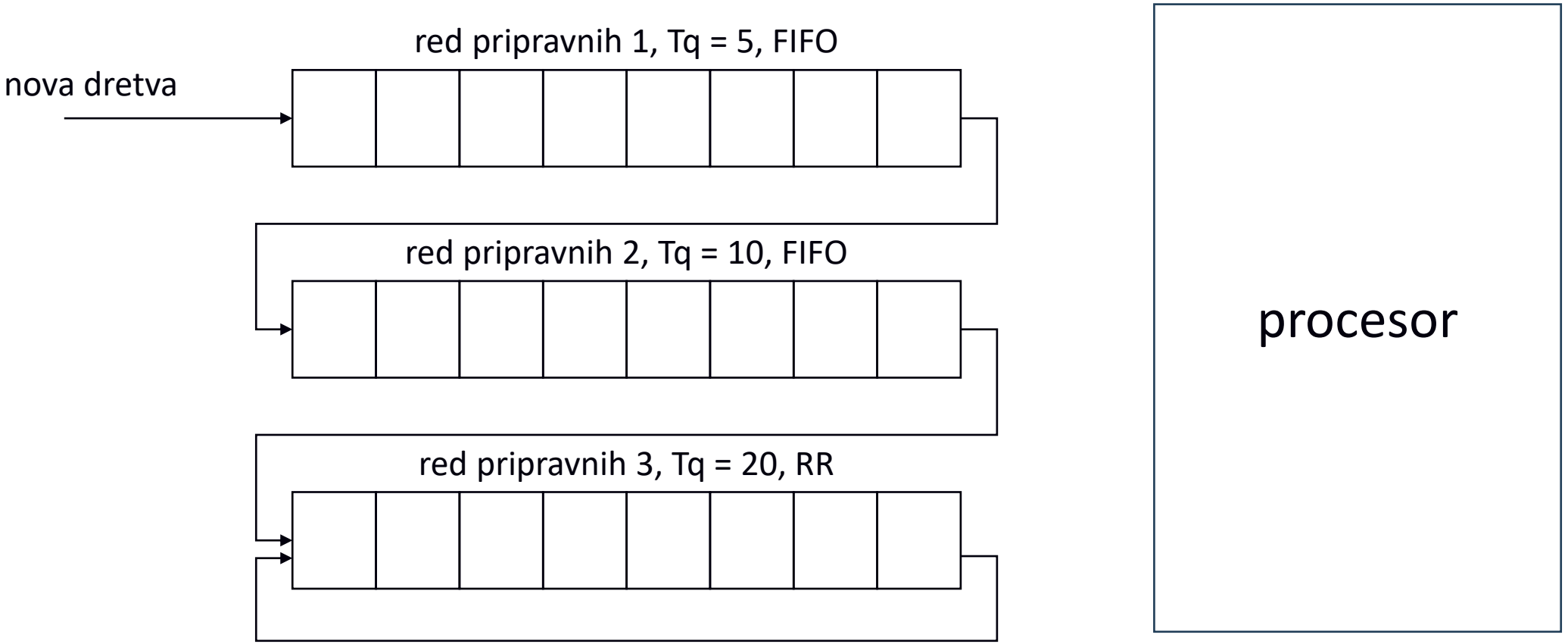
Raspoređivači za nekritične poslove

- osnovne ideje za svojstva raspoređivača
  - biti pravedan, svima isto (ili nekima više, ali prema postavkama)
  - pružiti maksimalno iskustvo korisniku
    - sučelje prema korisniku mora biti fluidno – brzo reagirati na njegove zahtjeve
    - to znači da reakcija na vanjske događaje bude brza, bez osjetnog pomaka
    - sučeljem upravljaju dretve koje uglavnom ništa ne rade osim u tim trenucima
      - stoga bi takve dretve trebalo favorizirati
      - čim postanu pripravne dati im procesor
      - ionako će se brzo maknuti je im je posao kratak
  - duge poslove po potrebi odgoditi
  - klasifikacija kratki/dugi obaviti algoritmom, a ne ručnim označavanjem
- teorijska strategija koja zadovoljava gornje ideje je MFQ

MFQ – multilevel feedback queue

- „višerazinsko raspoređivanje s povratnom vezom”
  - više FIFO redova, poredanih po prioritetu
  - kad nova dretva postane pripravna ulazi u najprioritetniji red, na kraj reda
  - aktivna dretva je prva iz prvog nepraznog najprioritetnijeg reda
  - aktivna dretva dobije kvant vremena
    - ako ta dretva završi prije isteka kvanta, onda izlazi iz sustava raspoređivača (nije više pripravna)
    - inače, kada potroši cijeli kvant, OS ju prekida, smanjuje prioritet za jedan te ju ubacuje u red pripravnih (na kraj prvog idućeg reda manjeg prioriteta)
  - dretve u redu najmanjeg prioriteta se poslužuju kružno (po trošenju kvanta vremena stavljaju se na kraj tog reda)
  - „kvant” vremena može imati različitu vrijednost, ovisno o prioritetu
    - npr. za veće prioritete je manji, a za manje veći

Višerazinsko raspoređivanje s povratnom vezom – primjer ostvarenja



Višerazinsko raspoređivanje s povratnom vezom – primjer rada

	C	T
D1	3 ms	50 ms
D2	10 ms	100 ms
D3	100 ms	2 s
D4	200 ms	5 s

Tq1 = 5
Tq2 = 10
Tq3 = 20

t=0

D1	D2	D3	D4
-	-	-	-
-	-	-	-

t=5

-	D1	D2	D3
-	-	-	D4
-	-	-	-

t=10

-	-	D1	D2
-	-	D3	D4
-	-	-	-

t=15

-	-	-	D1
-	D2	D3	D4
-	-	-	-

t=18

-	-	-	-
-	D2	D3	D4
-	-	-	-

t=28

-	-	-	-
-	-	D2	D3
-	-	-	D4

t=38

-	-	-	-
-	-	-	D2
-	-	D3	D4

t=43

-	-	-	-
-	-	-	-
-	-	D3	D4

t=50

-	-	-	D1
-	-	-	-
-	-	D3	D4

t=53

-	-	-	-
-	-	-	-
-	-	D3	D4

t=66

-	-	-	-
-	-	-	-
-	-	D4	D3

t=86

-	-	-	-
-	-	-	-
-	-	D4	D3

MFQ – multilevel feedback queue (2)

- MFQ se ne ostvaruje u operacijskim sustavima, ali drugim algoritmima nastoje simulirati prikazana svojstva
- npr. na Windowsima
  - prioritetno raspoređivanje s podjelom vremena, ali uz „iznimke”
    - najprioritetnija dretva dobiva najviše vremena (većinu), a ostale tek povremeno, kad OS detektira da gladuju
    - iako izgleda „čudno” dobro radi jer uglavnom dretve imaju isti prioritet
    - dretve čiji je proces u fokusu dobivaju povećanje prioriteta
- na Linuxu CFS – Completely Fair Scheduler
  - prati koliko je tko trebao dobiti a nije te na tome odlučuje o odabiru
    - dretva kojoj najviše duguje se izabire iduća
  - noviji EEVDF radi slično

9.2. Posebnosti višeprocesorskih sustava

- za svaki procesor zaseban red pripravnih dretvi
  - zbog performansi -- boljem korištenju priručnog spremnika
  - upravljanje priručnim spremnikom se odrađuje sklopovski, ali ipak to utječe na performanse
  - vrijedi općenito za većinu raspoređivača, za kritične i nekritične zadatke
- uzimanje u obzir različitosti procesora
  - utjecaj dijeljenja (nekih razina) priručnog spremnika jezgri u višejezgrenom procesoru
  - sklopovska višedretvenost (hyper-threading) povećava broj logičkih procesora s kojima OS upravlja, ali uzima li OS to u obzir, barem pri raspoređivanju kritičnih poslova?
  - brze jezgre (P-performanse), spore jezgre (E-efficiency)
  - NUMA

Raspoređivanje kritičnih poslova

- uglavnom se koristi prioritet kao kriterij odabira
- prioritetnije dretve uvijek prije one manjeg prioriteta
- kako to izvesti u višeprocesorskom sustavu gdje imamo više redova pripravnih dretvi?
  - koristi se **guranje** (push) i **povlačenje** (pull) dretvi
  - guranje: kad se neka dretva na nekom procesoru odblokira onda nju možda gurnuti nekom drugom procesoru koji izvodi dretvu manjeg prioriteta
  - povlačenje: kad se aktivna makne s nekog procesora (blokira, odgodi, završi i sl.), umjesto prve iz reda pripravnih tog procesora uzeti iz reda pripravnih nekog drugog procesora, ako ona tamo ima veći prioritet

Push/pull primjer

- P1:{6;3}, P2:{8;5,2}, P3:{9;4}, P4:{12;1} (aktivne i priprave po procesorima)

- P4 završi s dretvom 12; umjesto „svoje dretve 1” uzima dretvu 5 od P2
- P1:{6;3}, P2:{8;2}, P3:{9;4}, P4:{5;1}

- P3 odblokira dretve 7 i 10: gurne ih procesorima P4 i P1
- P1:{7;6,3}, P2:{8;2}, P3:{9;4}, P4:{10;5,1}
  - ili P3 zadrži dretvu 10 (da što prije ona nastavi), a dretve 9 i 7 gurne

Utjecaj manje prioritetnih dretvi na kritične

- korištenjem priručnog spremnika/sabirnice i manje prioritetnije dretve utječu na performanse prioritetnijih
  - agresivni rad (korištenje puno podataka) od strane nekritičnih dretvi
  - upravljanje priručnim spremnikom je sklopovski izvedeno
    - sklop ne zna za prioritete dretvi
- OS uglavnom ne intervenira zbog ovoga!
- ipak, kritična aplikacija bi se kroz OS mogla pobrinuti za takve neželjene situacije
  - npr. mogla bi stvoriti dodatne dretve većeg prioriteta od nekritičnih, koje samo zauzimaju procesor – privremeno izbacuju nekritične dok se neki kritični posao ne obavi do kraja

Raspoređivanje nekritičnih (običnih) dretvi

- osnovna načela su pravedna podjela vremena te što veća efikasnost
- nema potrebe za povlačenjem ili guranjem dretvi
- povremeno uravnotežiti opterećenja procesora – balansirati redove pripravnih dretvi
- raspoređivanje dretvi ili procesa?
  - kako gledati na dretve različitih procesa?
    - zanemariti pripadnost procesu i sve dretve smatrati ravnopravnima?
    - procesorsko vrijeme pravedno raspodijeliti procesima (tako da se gleda ukupno vrijeme svih dretvi pojedinog procesa)?
  - oba navedena pristupa imaju svoje prednosti i nedostatke: prvi je jednostavnije za ostvariti, a drugi je pravedniji

Optimiranje raspoređivanja za velike poslove

- neki procesi mogu stvoriti više dretvi koje paralelno rade na problemu
- međutim, takve dretve mogu imati povećanu potrebu za sinkronizacijom
- ako se istovremeno paralelno izvode na različitim procesorima mogu biti učinkovitije nego ako ne rade istovremeno – onda bi se mogle češće blokirati
- treba li OS to uzeti u obzir i pokušati takve dretve paralelno izvoditi na više procesora?
  - npr. grupno raspoređivanje (*gang scheduling*)
- nedostaci: složenost raspoređivača koji bi morao sinkronizirati sve procesore
- ovakav problem je uglavnom prisutan na poslužiteljima koji rade na zadacima različitih korisnika
  - oni to najčešće rješavaju slijednim izvođenjem poslova različitih korisnika
- na osobnim računalima (i sličnim sustavima) je najčešće samo jedan takav posao
  - sve ostale dretve sustava ukupno traže vrlo malo procesorskog vremena
  - npr. igre

Primjeri iz implementacije raspoređivanja u Linuxu

Raspoređivači ugrađeni u Linux

- Raspoređivači su podijeljeni u klase, pet ukupno, tri za korisničke dretve

1. STOP - interni, koristi se pri micanju svih zadataka s nekog procesora
2. DEADLINE – kritični zadaci, raspoređivanje prema krajnjem tr. završetka
   - SCHED_DEADLINE (ime u programima)
3. REALTIME – kritični zadaci, raspoređivanje prema prioritetu
   - SCHED_FIFO, SCHED_RR
4. FAIR – nekritični zadaci
   - SCHED_OTHER, SCHED_BATCH, SCHED_IDLE
5. IDLE – interni jezgrini zadaci najmanjeg prioriteta

- Pri odabiru aktivne dretve gornji raspoređivači se propituju tim redom

Odabir aktivne dretve

- dretva = zadatak; izvorno je u kodu „task”, ali je to zapravo dretva
- navedeni raspoređivači ostvaruju sučelje `sched_class`
- svi raspoređivači su u zajedničkoj listi prema navedenom redoslijedu
- pri odabiru aktivne dretve OS pita redom da li imaju dretvu za izvođenje
  - ako STOP nema, pita se DEADLINE, ako on nema onda REALTIME, pa FAIR te konačno IDLE (preko `pick_next_task` iz sučelja `sched_class`)
- svaki procesor ima svoj red pripravnih, za svaku klasu raspoređivača
- u višeprocesorskom sustavu ponekad se dretve prebacuju s reda jednog procesora u drugi (balansiranje opterećenja, prioriteti, ušteda energije, …)

Strukture podataka raspoređivača

- STOP i IDLE imaju samo jednu dretvu po procesoru
- DEADLINE ima prikladnu strukturu podataka (crveno-crno stablo) da brže odredi koju od svojih pripravnih dretvi odabrati
- REALTIME ima zaseban „red” za svaki prioritet
- FAIR uz CFS/EEVDF koristi crveno-crna stabla (jer se ono automatski uravnotežuje)
  - CFS je stariji, EEVDF je noviji (od 2024.)
  - koriste sličnu strukturu podataka

Osnovna načela pri raspoređivanju nekritičnih dretvi

- običnе/nekritične dretve, koriste se već opisane ideje što se očekuje
- parametar koji opisuje dretvu je „razina dobrote” – *nice level* (ili samo *nice*)
  - vrijednost za nice od -20 do +19, manja vrijednost veća dobrota
  - veća dobrota => veći udio procesorskog vremena
  - sve dretve dijele procesorsko vrijeme, ali one veće dobrote dobivaju više
    - razlika od jedne dobrote bi trebala predstavljati od 10 do 15 % više vremena
    - npr. razlika od pet dobrota: $1{,}15^5 \sim 2$ puta više vremena
  - negativne vrijednosti može postaviti samo root
  - pri normalnu pokretanju početna dobrota procesa je nula
  - pri radu OS može povećavati/smanjivati dobrotu ovisno o ponašanju dretve
    - ako ona puno radi (dugotrajni posao) dobrota joj se smanjuje (*nice* raste)
    - ako radi malo i rijetko onda joj dobrota raste (ali ne više od početne vrijednosti)

Algoritmi (info)

- U povijesti: O(1) – 2007. CFS (2.6.23) – 2023. EEVDF (6.6.)
- O(1) – složenost mu je bila O(1), ali je imao jako loša svojstva za interaktivne zadatke – puno se heuristike ubacivalo u raspoređivač da se to donekle popravi
- CFS – Potpuno pravedan raspoređivač – *completely fair scheduler* algoritamski je riješio (skoro sve) heuristike
  - problem koji se ipak ističe je loš odziv (latencija) kod nekih zadataka
  - logaritamska složenost (koriste se crveno-crna stabla)
- EEVDF – Raspoređivanje prema virtualnim rokovima
  - *earliest eligible virtual deadline first*
  - bolji odziv (zadaci koji se rijetko javljaju prije dođu na red)
  - logaritamska složenost (isto kao i CFS)

CFS – Potpuno pravedan raspoređivač (info)

- Osnovna ideja CFS-a jest da se procesorsko vrijeme pravedno podijeli svim pripravnim dretvama
- Slično kao i kod općeg raspoređivanja i raspoređivanja stablastom strukturom prikazanim u 4. poglavlju svaka bi dretva D_i trebala dobiti dio procesora - α_i
- naravno to nije moguće na taj način ostvariti
- vodi se evidencija koliko je koja dretva trebala dobiti vremena i koliko je dobila
  - razlika se koristi za uređenje dretvi u stablu
  - ona dretva koja ima najveću razliku (najviše joj sustav duguje procesorskog vremena) se prva izabire za izvođenje

CFS – Potpuno pravedan raspoređivač (2) (info)

- Primjer:
  - pet dretvi, sve iste razine dobrote
  - početno su jednake (isto im sustav duguje)
  - u idućem kvantu T svaka bi trebala dobiti T/5
  - ali kvant se dodjeljuje samo jednoj od njih, npr. D1
  - za D1 razlika između zasluženog i dobivenog: T/5 – T = -4/5 T
  - ostale: T/5 – 0 = T/5
  - sada je jedna od „ostalih” na prvom mjestu (D1 je na zadnjem)
    - obzirom da se koristi crveno-crno stablo „prvo mjesto” je najlijeviji čvor

Raspoređivanje prema virtualnim rokovima (info)

- algoritam prema članku iz 1995.: Ion Stoic, Hussein Abdel-Wahab: “Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation”.

1. ideja: pravedna podjela procesorskog vremena, ali uzimajući u obzir težine (npr. razinu dobrote)
2. svaka dretva daje zahtjev za procesorskim vremenom *r*
   - npr. to može biti kvant vremena koji ta dretva traži
3. algoritam proračunava do kada joj taj kvant treba biti dodijeljen – rok (deadline), uzimajući u obzir ostale dretve
4. algoritam također proračunava kad ta dretva može najranije dobiti procesorsko vrijeme uzimajući u obzir njeno prethodno izvođenje i izvođenje drugih dretvi (vrijeme treba pravedno podijeliti!)
   - trenutak podobnosti za izvođenje (*eligible*)

Raspoređivanje prema virtualnim rokovima (2) (info)

5. algoritam za aktivnu dretvu odabire jednu među *podobnima* čiji je rok najbliži
   - *Earliest Eligible Virtual Deadline First*

- ideja je da se preko *r* utječe na rok, da on za manji *r* bude prije
  - ali i dalje zadržavajući pravednu podjelu procesorskog vremena
  - dretve s manjim *r* prije dolaze na red (rok im je bliže)

- U nastavku slijedi malo detaljniji opis algoritma

EEVDF – 1 (info)

1. Svaka dretva D_i javlja se u sustav u trenutku t_0^i (kad je virtualno vrijeme bilo $V(t_0^i)$), ima svoju težinu w_i te se za nju prati koliko je procesorskog vremena dobila do trenutka t kroz $s_i(t_0^i, t)$.
2. Za raspoređivanje se razmatraju pripravne dretve označene skupom A.
3. Dretvi D_i u intervalu $[t_1, t_2)$ pripada procesorsko vrijeme $S_i(t_1, t_2)$:

$$S_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau.$$

Primjerice, kada bi N dretvi bilo pripravno u tom intervalu, sve s istom težinom w, svima bi pripadao jednak dio intervala, tj. $S_i(t_1, t_2) = \frac{t_2 - t_1}{N}$. Kada težine nisu iste, one dretve s većom težinom bi trebale dobiti više procesorskog vremena.

EEVDF – 2 (info)

4. Dug sustava prema dretvi D_i (engl. *lag*) definira se s:

   $lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t)$.

5. Kada je $lag_i(t) < 0$ dretva D_i je do t dobila više nego što je trebala.

   Npr. u $t = 0$ neka dretva dobije kvant vremena od 5 ms. U $t = 5$ ms ona je dobila više nego je trebala, jer je u teoriji to vrijeme trebalo ravnomjerno podijeliti svima.

6. Pri raspoređivanju se gledaju samo one dretve s pozitivnim dugom (kojima sustav duguje), tj. definira se da su takve dretve podobne za izvođenje (engl. *eligible*), spadaju u skup A.

EEVDF – 3 (info)

7. Raspoređivač za odabir aktivne dretve koristi:
   - $Ve_i(t)$ – virtualni trenutak kad dretva D_i postaje podobna (kada vrijedi $lag_i(t) = 0$)
   - $r_i(t)$ – potrebno procesorsko vrijeme (preostali kvant vremena za dretvu D_i) te
   - $Vd_i(t)$ – rok (engl. *virtual deadline*) kad bi dretva D_i trebala biti gotova, uz njezin udio u procesorskom vremenu prema težinama.
8. $Ve_i(t)$ za dretvu D_i se računa prema prema:

   $$Ve_i(t) = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i},$$

   tj. kao suma virtualnog vremena kad se dretva pojavila u sustavu i već dobivenog procesorskog vremena skaliranog s težinom dretve.
9. Vremena $Ve_i(t)$, $Vd_i(t)$ i $V(t_0^i)$ su virtualna, dok t, t_0^i, s_i, S_i i lag_i to nisu.

EEVDF – 4 (info)

10. Virtualna vremena $V(t)$ se računaju prema:

$$V(t) = \int_0^t \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau$$

tj. stvarno vrijeme skalirano je težinama – što imamo više dretvi virtualno vrijeme teče sporije.

11. $Vd_i(t)$ za dretvu D_i se računa prema:

$$Vd_i(t) = Ve_i(t) + \frac{r_i(t)}{w_i},$$

tj. od $Ve_i(t)$ do budućeg virtualnog trenutka kad bi dretva trebala biti gotova obzirom na zahtjev $r_i(t)$ i njezin udio u procesorskom vremenu ($r_i(t)$ skaliran sa w_i).

EEVDF – 5 (info)

12. Raspoređivač odabire dretvu s najmanjim $Vd_i(t)$ (najbliži rok) za koji vrijedi $Ve_i(t) \leq V(t)$ (među svim dretvama podobnim za izvođenje).

Što je dretvi D_i kvant r_i manji to će i virtualni rok Vd_i biti bliži te će dretva prije doći na red za izvođenje. Ovo je potrebno dretvama koje trebaju nakon buđenja (ili odblokiranja) što prije dobiti procesorsko vrijeme (što manju latenciju). Uz ovaj algoritam latencija se smanjuje bez nadodate heuristike koja je potrebna kada se primjenjivao CFS.

CFS – crveno-crno stablo

- iako se koristi stablo koristi se naziv *red*, *run queue*, kratica *rq*
- struktura `cfs_rq` opisuje stablo
- svaki procesor ima zasebno stablo
- elementi stabla (čvorovi) su strukture `sched_entity`, koji predstavlja
  - dretvu opisanu kroz `task_struct`
  - grupu dretvi opisanu kroz – `task_group`
- zapravo se `sched_entity` ugrađuje u `task_struct/task_group`
- `sched_entity` reprezentira dretvu ili grupu unutar stabla
  - kad bude odabran za izvođenje onda ta dretva postaje aktivna ili ako je to grupa dretvi, onda se rekurzivno iz stabla grupe odabire jedna dretva
    - `task_group` ima novo stablo za dretve koje sadrži

Grupe zadataka – `task_group`

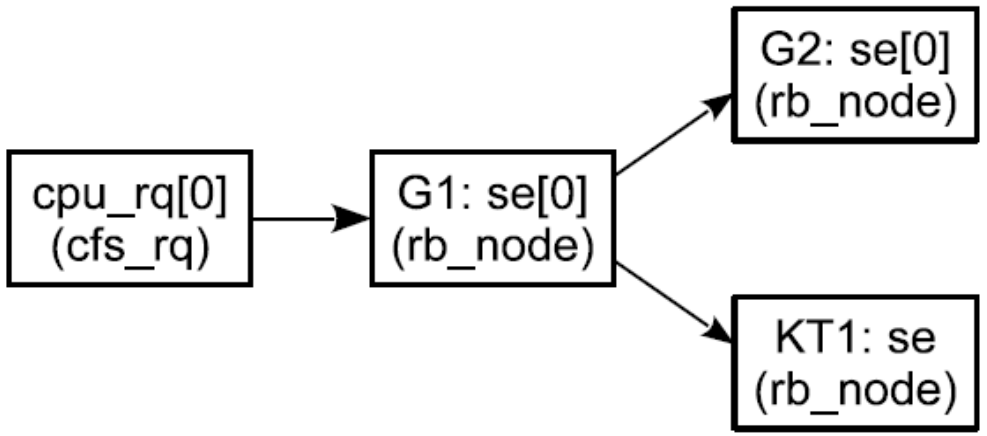
- mehanizam grupiranja omogućuje dodatne mogućnosti, npr.
  - grupiranje dretvi jednog procesa (npr. radi pravednije podjele vremena)
  - optimiranje raspoređivanja, paralelno izvođenje zadataka, …
  - NUMA sustavi i raspodjela dretvi na grozdove i sl.
- stvaranje grupa može biti ručno ili automatski
  - npr. svako pokretanje programa nova grupa (i *session*)
- `task_group` ima/može imati po jedan `sched_entity` za svaki procesor
  - svaki takav `sched_entity` ima zaseban red (`cfs_rq`) za dretve ili podgrupe
    - u tom redu su ili samo zadaci ili samo grupe
    - iznimka je samo početni red u kojem mogu biti i jezgrine dretve a ne samo grupe

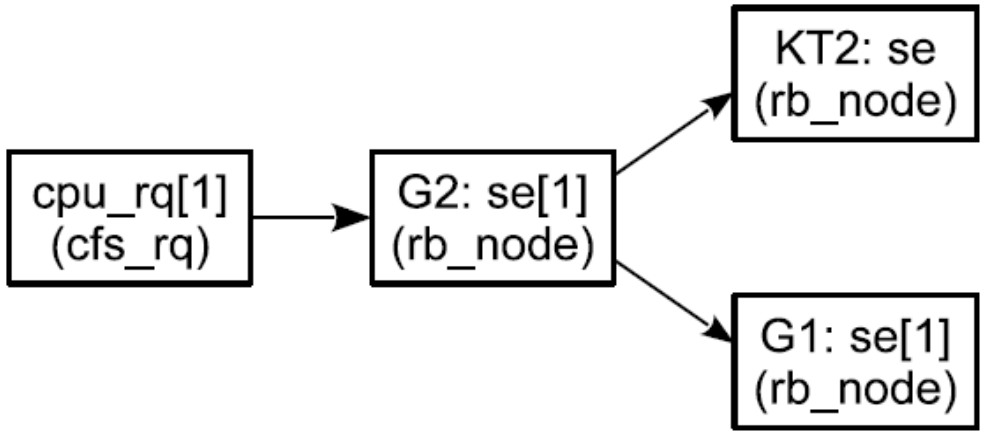
Izvadak iz struktura podataka

```
struct cfs_rq - sadrži crveno-crno stablo
    struct rb_root_cached tasks_timeline; - pokazuje na početni čvor u tom stablu

struct sched_entity - jedan čvor u 'tasks_timeline'
    struct rb_node run_node; - za ostvarenje stabla
    struct cfs_rq *my_q;     - novi red, ako opisuje grupu

struct task_stuct - opisuje zadatak, sadrži (pored ostalog)
    struct sched_entity se;

struct task_group -- opisuje grupu zadataka ili podgrupa, sadrži (pored ostalog)
    struct sched_entity **se; - za svaki procesor jedan se (polje kazaljki)
    komentar u kodu: schedulable entities of this group on each CPU

    struct cfs_rq **cfs_rq; -- za svaki procesor jedan (my_q od sched_entity)
    komentar u kodu: runqueue "owned" by this group on each CPU
```

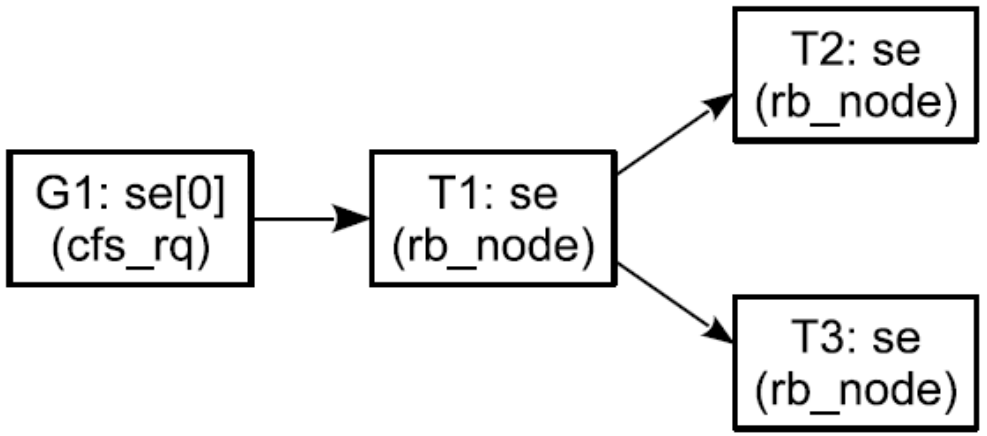
Primjer strukture podataka

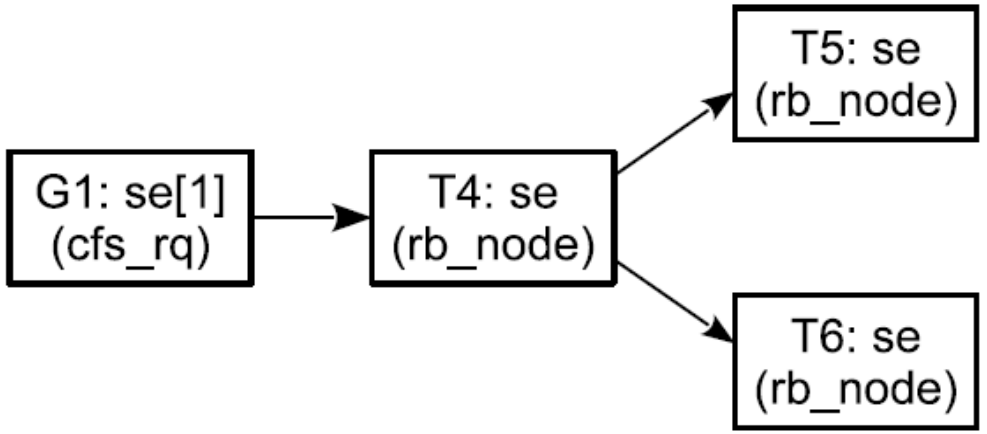
crveno-crno stablo preko elementa **`rb_node`** strukture **`sched_entity`**





crveno-crno stablo preko elementa **`rb_node`** strukture **`sched_entity`**





Oznake: G-grupa, KT-jezgrin zadatak, T-zadatak, se-element **`sched_entity`**

Procesorske domene (scheduling domain)

- ideja: bolje iskoristiti procesore ako se zna njihova hijerarhija
  - ali mogu i neke druge stvari, kao particioniranje sustava, procesa…
- svaka domena `struct sched_domain` sadrži skup procesora nekih zajedničkih svojstava
- domene mogu biti hijerarhijski povezane
- svaka domena ima jednu ili više grupa zadataka (prethodno opisan mehanizam) koje se raspoređuju nad pripravnim procesorima

Procesorske domene (2)

- za svaku domenu definira se skup pravila kojima se nastoji iskoristiti svojstva tih procesora
  - kako često raditi balansiranje među redovima procesora (npr. za sklopovsku višedretvenost vrlo često, višejezgrene procesore nešto rjeđe, NUMA sustave još rjeđe)
  - koliko dugo vremena se neki zadatak može ne izvoditi na nekom procesoru, a da ga kasnije i dalje ima smisla vratiti na isti procesor zbog mogućih podataka u priručnom spremniku
  - dijeljenje napajanja – može li se samo jedan ugasiti ili slično?

Raspoređivanje u operacijskim sustavima Microsoft Windows

Osnovne stvari o raspoređivanju

Novosti uz Windows 11 i novije generacije procesora s P i E jezgrama

Raspoređivanje dretvi – prema prioritetu

- prioritet se postavlja preko
  - prioritetne klase procesa
    - IDLE_PRIORITY_CLASS, BELOW_NORMAL*, NORMAL*, ABOVE_NORMAL*, HIGH*, REALTIME* (sučelje: `SetPriorityClass(hProcess, dwPriorityClass)`)
  - prioritetne razine dretve
    - THREAD_PRIORITY_IDLE, *LOWEST, *BELOW_NORMAL, *NORMAL, *ABOVE_NORMAL, *HIGHEST, *TIME_CRITICAL (sučelje: `SetThreadPriority(hThread, nPriority)`)
- kritične dretve u klasi REALTIME_PRIORITY_CLASS imaju prioritet 16-31, ostale (obične) 0-15
- kritičnim dretvama OS ne mijenja prioritet, koristi podjelu vremena za dretve istog prioriteta
- ako su svi procesori zauzeti kritičnim dretvama, obične čekaju

Raspoređivanje dretvi – obične dretve

- obične dretve se također raspoređuju prema prioritetu, ali uz iznimke
- običnim dretvama OS može promijeniti prioritet
  - kada aplikacije dođe u fokus korisnika, njenim dretvama se poveća prioritet
  - kada aplikacija ode u pozadinu, vrati joj se prijašnji prioritet
  - povremeno (npr. svake sekunde) raspoređivač provjerava ima li dretvi koje dugo nisu dobile procesorsko vrijeme – takvim dretvama privremeno povećava prioritet (možda ne svima uvijek, ako ih ima puno)
  - radi osiguranja kvalitete usluge, postoji servis koji multimedijalnim aplikacijama po potrebi povećava prioritet
- (ukratko) ako programi nisu drukčije tražili, aplikacija u fokusu dobiva najveći prioritet i ako ona treba puno procesorskog vremena dobiti će ga (a ostale dretve malo) – to je uglavnom ono što i (jedini) korisnik traži

Višeprocesorski sustavi

- za svaki procesor (jezgru) postoji zaseban „red pripravnih”
- dretve se mogu rasporediti na bilo koji procesor
  - ipak, nastoji se koristiti onaj gdje su prethodno bile (zbog priručna spremnika)
- *afinitet*: dretvama se mogu postaviti i ograničenja na kojim se procesorima mogu izvoditi
  - raspoređivač to mora poštivati
  - `SetProcessAffinityMask(hProcess, AffinityMask);`
- *idealan procesor:* dretvi se može postaviti njen „idealan” procesor
  - raspoređivač nastoji dretvu staviti na taj procesor, ako može
  - `SetThreadIdealProcessor(hThread, IdealProcessor);`

Mnogojezgreni procesori i procesorske grupe

- u sustavima s puno jezgri (64+) Windowsi uvode Procesorske Grupe
  - dretve nekog procesa mogu biti samo u jednoj grupi (prije Window 11)
  - dretve nekog procesa mogu biti u bilo kojoj grupi, tj. različite dretve mogu biti u različitim grupama, ali mogu i u istoj (Windows 11+)
- posebno zanimljivo za NUMA sustave
  - npr. za grupiranje dretvi jednog procesa na jednom čvoru

Heterogeni procesori (P+E), Intel Thread Director

- Novije generacije Intelovih procesora (serija 12+) imaju na čipu i *Intel Thread Director*
  - njegova je zadaća da prati rad dretvi te da na osnovu onoga što rade (kakve instrukcije izvode) predlaže OS-u (Windowsima 11+) kako da raspoređuje te dretve po jezgrama
  - svrstava dretve u klase: prioritetni zadaci, zadaci u pozadini, AI zadaci
  - uzima u obzir trenutnu potrošnju procesora (TDP), način rada (*operating condition*) te postavke sustava (*power settings*)
  - Windows 11 može komunicirati s tim dijelom procesora
  - OS može koristiti te podatke da bolje rasporedi dretve (ali i ne mora)

Heterogeni procesori (P+E), Windows 11

- Windows 11 je „svjestan” razlika između P (*Performance core*) i E (*Efficiency core*) jezgri i nastoji ih pravilno koristiti
- ideja bi bila da poslove koji nisu hitni prebaci na E jezgre
- ali kako raspoređivač zaključuje da neki poslovi nisu hitni?
  - prema mnogim raspravama na forumima izgleda da čim se proces stavi van fokusa korisnika (u pozadinu, tj. u fokus se stavi neki drugi program), tada raspoređivač ovakav proces prebacuje na E jezgre
  - mnogi se žale na to jer su te jezgre sporije, a oni žele da njihov zahtjevan program (simulacija, proračun, kompresija ili slično) koristi brze jezgre dok oni nešto drugo (nezahtjevno) rade na istom računalu
    - P jezgre su slobodne a ne koriste se, iako je sustav u high-performance načinu rada

Heterogeni procesori (P+E), Windows 11 (2)

- kako riješiti problem „automatskog” prebacivanja na E jezgre?
  - ostaviti program u fokusu – to nije ono što korisnici žele (aktivno čekati)
  - postaviti afinitet procesu na samo P jezgre – ali onda neće koristiti i E jezgre, a zašto ne?
- možda je u međuvremenu problem riješen ili se radi na tome …

Windows 11 + *Efficiency mode*

- *Task Manager* na Windows 11 OS-u (uz 22H2 ažuriranje) ima novu mogućnost postavljanja nekog procesa u način rada *Efficiency mode* (ili isključivanja tog načina za neki proces)
  - time se proces označava kao pozadinski, smanjuje mu se prioritet
  - ideje ovog mehanizma su:
    - smanjiti potrošnju – bitno općenito, posebice na uređajima na baterijama
    - povećati odziv drugih aplikacija (koje nisu u pozadini)
  - više na:
    - https://devblogs.microsoft.com/performance-diagnostics/reduce-process-interference-with-task-manager-efficiency-mode/
    - https://devblogs.microsoft.com/performance-diagnostics/introducing-ecoqos/
- aplikacije mogu tražiti korištenje tog načina
  - i to rade, što ljuti mnoge korisnike koji traže kako to isključiti …