Paralelno programiranje

Današnja tehnologija izgradnje procesora dosegla je gotovo svoje vrhunce. Naime, razvoj procesora vođen je povećanjem frekvencija rada procesora, što se je postizalo ugradnjom većeg broja tranzistora na sve manje čipove.

Gordon Moore (jedan od utemeljitelja tvrtke Intel) izjavio je u časopisu Electronics iz 19. travnja 1965: „Broj tranzistora koji se po najpovoljnijoj cijeni mogu smjestiti na čip udvostručava se otprilike svake dvije godine“. Ova njegova rečenica poznata je još i kao Mooreov zakon. Međutim, zbog povećavanja broja tranzistora povećava se utrošak energije a time i zagrijavanje procesora. Dakle, povećanje broja tranzistora u nekom će trenutku doći do kritične brojke i neće više biti moguće ugrađivati dodatne tranzistore, čime ćemo dosegnuti vrhunac brzine procesora na ovoj tehnologiji.

Alternativa broju tranzistora na čipu je ugradnja više procesora u računalo. Ugradnja više procesora ima i svoje nedostatke (brzina komunikacije među procesorima, potrošnja energije,…).

Druga alternativa je izgradnja procesora kod kojih će ne jednom čipu biti ugrađeno više procesorskih jezgri.

Tako današnja računala najčešće imaju dvije, četiri ili više jezgri koje mogu neovisno izvoditi različite zadatke ali isto tako mogu zajednički izvoditi jedan isti zadatak.

Nas će osobito zanimati ideja da dvije ili više jezgri izvode isti zadatak. Ilustrirajmo to jednim jednostavnim primjerom:

Pretpostavimo da treba prepisati tekst od 20 stranica. Jednoj osobi bi za to primjerice trebalo 100 minuta. Ukoliko bi primjerice tekst podijelili na 4 osobe, i svakoj dali po 5 stranica teksta, za pretpostavljati je da bi posao bio gotov za oko 25 minuta, ukoliko sve osobe tipkaju otprilike istom brzinom.

Za očekivati je da se slično događa i u računalu. Primjerice, ukoliko neki zahtjevni posao izvodimo na jednoj jezgri računala onda bi za to trebalo vjerojatno više vremena nego ukoliko ga izvodimo na više jezgri istog računala.

U nastavku ćemo se u to pokušati i uvjeriti.

Dretve i procesi

Program zapisan u nekom programskom jeziku čini niz naredbi zapisanih tako da ga računalo može izvoditi. Kada neki program pokrenemo na računalu, taj niz naredbi se smjeti u radni spremnik računala. Osim toga svaki program ima i neke svoje varijable te se u radnoj memoriji pokretanjem programa alocira i memorija za sadržaje tih varijabli. Svojim pokretanjem program dobiva još neka nova obilježja (vrijeme početka i završetka programa, trajanje programa, i sl.). Program u trenutku izvođenja zvat ćemo **proces**. U računalu se istovremeno može odvijati više procesa. Svaki od njih ima svoj dio memorijskog prostora kako za same naredbe tako i alocirani prostor za varijable i sl. Primjerice ako na računalu pokrenemo dva puta program za crtanje (Paint), kreirat će se dva neovisna procesa, svaki će dobiti svoj dio prostora u RAM-u itd. Ovi procesi se naizgled izvode paralelno. Takvo „paralelno“ izvođenje više procesa omogućava operacijski sustav računala koji za svaki proces rezervira hardverske resurse. Operacijski sustav to najčešće realizira tzv. **kružnom podjelom vremena** (eng.

Round Robin), koja se temelji na podjeli procesorskog vremena (eng. *Time Sharing*). Pri takvoj podjeli vremena svi aktivni procesi dobiju dio procesorskog vremena koji se izvode, nakon tog dijela izvodi se drugi proces, zatim treći itd., sve do posljednjeg procesa, nakon kojeg se ponovo izvodi prvi proces itd. Ukoliko imamo više jezgri procesora događa se slična stvar, neki procesi izvodit će se na jednoj, neki drugi na drugoj jezgri itd.

Pri izvođenju nekog procesa procesor izvodi niz instrukcija – **instrukcijsku dretvu** odnosno samo **dretvu**. Valja razlikovati programski niz naredbi od dretve. Naime programski niz naredbi je niz naredbi onakvih kakvi su zapisane u spremniku računala, dok dretvu čini niz naredbi koje su povezane vremenskim slijedom.

Jedan proces ima najmanje jednu a može imati i više dretvi koje se mogu izvoditi paralelno ili prividno paralelno (svaka se može izvoditi na svojoj jezgri procesora ili se sve mogu izvoditi na istoj jezgri).

U primjerima iz prethodnog poglavlja imali smo više dretvi unutar istog procesa (Poslužitelj). Za razliku od procesa, gdje svaki proces ima svoj memorijski prostor, sve dretve nekog procesa dijele isti memorijski prostor.

Slično kao što smo u 1. poglavlju program dijelili na dretve na analogan način je jedan program moguće podijeliti i na više procesa.

Procesi u programskom jeziku Python

Za kreiranje aplikacija unutar kojih će se paralelno izvoditi više procesa koristit ćemo Pythonov modul **`multiprocessing`**. Osnovna klasa ovog modula je klasa **Process**, koja je u osnovi vrlo slična klasi **`Thread`**.

Osim klase **Process** modul **multiprocessing** ima još i neke funkcije. Jedna od korisnih funkcija bit će funkcija `cpu_count()`. Ova funkcija vraća broj jezgri procesora. Ilustrirajmo uporabu ove funkcije na nekom računalu:

```
>>> from multiprocessing import *
>>> cpu_count()
4
```

Konstruktor klase **Process** slično kao i konstruktor klase **Thread** ima dva parametra:

```
Process(target = ime_funkcije, args = (arg1, arg2,…))
```

Neka od svojstava klase **Process** dana su u sljedećoj tablici:

Tablica 1: Najčešće korištena svojstva i metode klase **Process**

Naziv	Opis
`join([t])`	čeka da proces završi. Ukoliko je postavljen opcionalni parametar *t* proces će biti prekinut nakon isteka vremena *t*
`name`	vraća naziv procesa
`is_alive()`	vraća *True* ako je proces aktivan a *False* inače
`pid`	vraća ID procesa
`Terminate()`	prekida izvođenje procesa

Još jedna zgodna funkcija je `current_process()` koja vraća objekt tipa `Process` a koji predstavlja trenutni proces.

Svaki proces u računalu ima svoje ime te jedinstveni ID.

Primjer 5:

Napišimo program koji će pokretati 4 procesa. Svaki proces prilikom svog pokretanja treba ispisati poruku svoje ime i ID. Nadalje proces treba zaustaviti na nekoliko (random) sekundi te ponovo ispisati ID procesa te vrijeme koje je proces bio zaustavljen.

Rješenje:

Funkcija koju će svaki proces izvoditi je oblika:

```
def f():
    p = current_process()
    print('ID: {0} ime: {1}'.format(p.pid, p.name))
    t = randint(1, 10)
    sleep(t)
    print('Proces {0} je završio nakon {1}s'.format(p.pid, t))
    return
```

dok će glavni dio programa koji će pokrenuti 4 procesa te potom ispisivati nazive procesa i još jednom ID-eve procesa je sljedeći:

```
if __name__ == '__main__':
    procesi = [Process(target = f) for i in range(4)]
    for p in procesi:
        p.start()
    for p in procesi:
        p.join()
```

Pokretanjem ovog programa u komandnom promptu dobit ćemo ispis nalik sljedećem:

```
ID: 10784 ime: Process-1
ID: 11708 ime: Process-2
ID: 11604 ime: Process-4
ID: 11864 ime: Process-3
Proces 10784 je završio nakon 3s
Proces 11864 je završio nakon 4s
Proces 11604 je završio nakon 6s
Proces 11708 je završio nakon 10s
```

Procesu je prilikom pokretanja moguće dodijeliti i ime. Ime se navodi kao parametar konstruktora klase **`Process`**. Radi se o parametru `name`.

Izmijenimo prethodni primjer tako da prilikom kreiranja procesu dodijelimo i ime koje će biti oblika *P-[i]*, pri čemu će *[i]* biti redni broj procesa. Razlika će biti samo kod kreiranja liste procesa:

```
procesi = [Process(target = f, name = 'P-' + str(i + 1)) for i in range(4)]
```

Nakon što smo naučili osnovno o procesima uvjerimo se da će paralelno izvođenje procesa koji rješavaju neki problem ubrzati izvođenje tog procesa.

Primjer 6:

Napišimo program koji će pokretati prebrojiti sve proste brojeve do 1000000 te će izmjeriti vrijeme izvođenja tog programa.

Rješenje:

S problemom iz ovog primjera smo se susretali više puta te njegovo rješenje nećemo specijalno objašnjavati:

```
from time import *

def prost(n):
    for i in range(2, round(n ** 0.5 + 1)):
        if n % i == 0:
            return False
    return True

def prosti(a, b):
    t = 0
    for i in range(a, b):
        if prost(i):
            t += 1
    return t

if __name__ == '__main__':
    a = 2
    b = 1000001
    poc = time()
    print(prosti(a, b))
    kraj = time()
    print('{:.5f}'.format(kraj - poc))
```

Izvođenje ovog programa na nekoj arhitekturi računala rezultirat će sljedećim ispisom:

```
78498
12.12407
```

To znači da postoji 78498 prostih brojeva manjih od 1000000 i vrijeme izvođenja ovog programa trajalo je 12.12407s.

Pokušajmo posao brojanja bojeva do 1000000 podijeliti tako da se može odvijati paralelno kao više procesa. Nameće se pomisao da bismo to mogli načiniti da kreiramo primjerice dva procesa, prvi proces bi brojao proste brojeve od 2 do 500000 a drugi od 500001 do 1000000. Primijetimo da za rješenje ovog problema možemo koristiti rješenje prethodnog primjera na način da će prvi proces funkciju *prosti* pokrenuti s parametrima *(2, 500000)* a drugi će funkciju *prosti* okrenuti s parametrima *(500000, 1000001)*.

U tu svrhu ćemo izmijeniti glavni program:

```
if __name__ == '__main__':
    a = 2
    b = 1000001
    poc = time()

    p1 = Process(target = prosti, args = (a, b // 2))
    p1.start()
```

```
    p2 = Process(target = prosti, args = (b // 2 + 1, b))
    p2.start()

    p1.join()
    p2.join()

    kraj = time()
    print('{:.5f}'.format(kraj - poc))
```

Izvođenje ovog programa na istoj arhitekturi računala rezultirat će sljedećim ispisom:

```
7.92392
```

Dobiveni ispis predstavlja trajanje brojanje prostih brojeva do 1000000 ako program pokrenemo kao dva procesa od kojih svaki broji proste brojeve unutar jednog segmenta.

Kao što možemo primijetiti dobiveno trajanje je znatno kraće od trajanja prvog izvođenja prvog programa. Moglo bi nas zanimati zašto to vrijeme nije upola kraća od prvog dobivenog vremena. Razloga za to je nekoliko. Neki od njih su:

- računalo neće uvijek istom brzinom izvoditi ove programe. Stoga ćemo gotovo za svako pokretanje programa dobivati različita vremena. Radi se o tome da osim ovih, naših procesa, računalo izvodi i još neke druge procese a ovisno o zahtjevnosti tih drugih procesa računalo je dodijelilo vrijeme našim procesima
- intervale za provjeru prostih brojeva nismo jednoliko podijelili. Naime, znatno je teže provjeriti je li primjerice broj 789533 prost nego je li prost broj 11
- računalu je potrebno neko vrijeme da kreira svaki od procesa
- itd.

Primijetimo da kod ovog izvođenja programa nismo dobili ukupni broj prostih brojeva. Naime, svaki proces je pokrenuo funkciju s određenim parametrima ali proces ne može dobiti rezultat koji funkcija vraća. Tom problemu ćemo se vratiti kasnije.

U našem primjeru smo rješenje realizirali tako da radi za dva procesa i imali bismo malo posla ako bismo ga htjeli realizirati da radi za neki veći broj procesa. Pokušajmo izmijeniti naš program na način da se on relativno može prilagoditi da radi za proizvoljan (n) broj procesa:

```
if __name__ == '__main__':
    a = 2
    b = 1000002
    n = 2
    d = (b - a) // n
    poc = time()

    p = [Process(target = prosti, args = (a + i * d, a + (i + 1) * d)) for i in
            range(n)]

    for t in p:
        t.start()

    for t in p:
        t.join()

    kraj = time()
    print('{:.5f}'.format(kraj - poc))
```

Broj procesa programa definiran je varijablom n. Kod izvođenja programa poželjno je voditi računa da je kvocijent *(b - a) / n* prirodan broj kako ne bi došlo do preskakanja nekih brojeva kod definiranja intervala (parametara funkcije).

Pokrenemo li primjerice program s vrijednošću parametra *n = 4* na istoj arhitekturi računala dobit ćemo trajanje oblika *5.63488s*, što je još bolje od *7.92392s* za dva procesa.

Pokretanjem programa za *n = 8* procesa dobivamo trajanje: *3.74377s*; za *n = 16* trajanje od *3.99361s*,… Da se zaključiti da trajanje za *n = 16* ne pridonosi više brzini. Zanimljivo je primijetiti da računalo na kojem se odvija testiranje ima 4 jezgre i da se optimalno izvođenje postiže kada se program izvodi kao 8-procesorski.

Sinkronizacija paralelnih procesa

U prethodnom primjeru osnovni cilj nam je bio izmjeriti vrijeme trajanja paralelnih procesa a kod problema brojanja prostih brojeva unutar zadanog intervala. Uvjerili smo se da ukoliko smo isti program pokrenuli kao jedan proces koji je trebao brojati sve proste brojeve unutar intervala od 2 do 1000000 onda je taj proces trajao znatno duže nego ako smo primjerice pokrenuli 4 procesa od kojih je svaki brojao proste brojeve na četvrtini danog intervala. Kada smo problem pustili da se izvodi kao 4 procesa na kraju nismo uspjeli dobiti ukupni broj prostih brojeva do 1000000. U nastavku ćemo se pozabaviti upravo time. Htjet ćemo na neki način da nakon što se tih primjerice 4 procesa završi da ipak znamo koliko su oni prostih brojeva ukupno izbrojali.

Rekli smo da svaki proces koji se pokrene ima između ostaloga svoj memorijski prostor. Međutim, moguće je za sve procese definirati i neke zajedničke memorijske lokacije. Upravo ćemo se time baviti u nastavku.

Podaci se u zajedničke lokacije mogu pisati koristeći objekte tipa **Value** i **Array**. Objekt tipa Value kreirat ćemo na sljedeći način:

`ime_objekta = Value('i', 0)` odnosno

`ime_objekta = Value('d', 0.0)`.

U prvom slučaju kreirat ćemo objekt u kojega ćemo upisivati cjelobrojne vrijednosti dok ćemo u drugom slučaju kreirati objekt u kojeg ćemo upisivati realne vrijednosti. U oba slučaja je početna vrijednost objekta postavljena na 0.

Naredbom:

`ime_objekta = Array('i', lista)` kreirat ćemo objekt tipa array, pri čemu će elementi polja biti cjelobrojnog tipa ('i') dok će početne vrijednosti polja biti zadane listom **`lista`**.

Riješimo sada primjer brojanja prostih brojeva koristeći objekt tipa *Value*. Vrijednost objekta *o* tipa *Value* dobit ćemo naredbom: **`o.value`**. Objekt tipa *Value* morat će biti parametar funkcije koja se izvodi unutar procesa.

Samo po sebi se nameće da bismo objekt tipa *Value* mogli koristiti za brojanje prostih brojeva u svim procesima. Bilo koji proces kada naiđe na prosti broj povećat će vrijednost ove varijable za 1 (ona će zamijeniti lokalnu varijablu *t*). Funkcija *prosti* sada će imati oblik:

```
def prosti(a, b, t):
    for i in range(a, b):
        if prost(i):
            t.value += 1
    return
```

Dok će dio u kojem ćemo kreirati procese imati oblik:

```
brojac = Value('i', 0)
p = [Process(target = prosti, args = (a + i * d, a + (i + 1) * d, brojac)) for i in
          range(n)]
```

Te ćemo na kraju ispisati vrijednost bojača:

```
print('Ukupno brojeva: {}'.format(brojac.value))
```

Nakon izvođenja programa primijetit ćemo da ukupni broj prostih brojeva nije isti kao kad smo program pokrenuli na samom početku poglavlja. Prisjetimo se u tom slučaju je ukupni broj prostih brojeva bio 78498. Dobiveni broj u ovom slučaju je znatno manji i neće biti isti prilikom svakog pokretanja programa. Razlog ovoj razlici leži u sljedećem: operacija povećanja sadržaja varijable je kompleksna operacija i izvodi se u nekoliko koraka, od kojih su neki:

1. učitaj vrijednost varijable
2. pohrani vrijednost varijable u lokalni spremnik
3. povećaj vrijednost lokalnog spremnika za 1
4. pohrani sadržaj lokalnog spremnika na memorijsku lokaciju od kuda je vrijednost pročitana u 1. koraku.

Budući da se ovdje radi o paralelnim procesima može se dogoditi da dva ili više procesa istovremeno pristupaju ovoj varijabli i tu može doći do neželjenih posljedica.

Ilustrirajmo to:

Pretpostavimo da u memorijskoj lokaciji `brojac` piše vrijednost 12. Ilustrirajmo problem na dva procesa.

- prvi proces čita sadržaj varijable *brojac* (12) i sprema ga u svoj lokalni spremnik
- prvi proces povećava vrijednost svog lokalnog spremnika (13)
- drugi proces čita sadržaj varijable *brojac* (12) i sprema ga u svoj lokalni spremnik
- prvi proces piše vrijednost iz svog lokalnog spremnika u varijablu *brojac* (13)
- drugi proces povećava sadržaj lokalnog spremnika (13)
- drugi proces piše vrijednost iz svog lokalnog spremnika u varijablu *brojac* (13)

nakon ovih koraka vrijednost varijable *brojac* povećana je za 1 iako su oba procesa trebala povećavati vrijednost za 1 odnosno, vrijednost varijable *brojac* je trebala biti povećana ukupno za 2.

Kako bismo izbjegli ovakav scenarij potrebno je na neki način omogućiti da dok prvi proces povećava sadržaj varijable *brojac* niti jedan drugi proces ne smije pristupiti sadržaju varijable *brojac*. To ćemo načiniti tako da za vrijeme povećavanja sadržaja varijable „zaključamo“ tu varijablu. To ćemo načiniti

koristeći objekt tipa `Lock()` koji omogućava „zaključavanje“ memorije na neko vrijeme. Zaključavanje se realizira metodama:

- `acquire()`
- `release()`

Objekt tipa Lock kreirat ćemo u glavnom programu i to će također biti parametri funkcije koja se izvodi kao funkcija procesa.

Funkcija će sada imati oblik:

```
def prosti(a, b, t, l):
    for i in range(a, b):
        if prost(i):
            l.acquire()
            t.value += 1
            l.release()
    return
```

Dok će glavni program imati oblik:

```
l = Lock()
p = [Process(target = prosti, args = (a + i * d, a + (i + 1) * d, brojac, l)) for i
          in range(n)]
```

Nakon ovoga će program ispisivati ispravno rješenje.

"Bazen" poslova

Izmijenimo funkciju `prosti()` na način da proces ispišemo poruku prije i nakon petlje:

```
def prosti(a, b, t, l):
    p = current_process()
    print('Proces {} je započeo'.format(p.name))
    for i in range(a, b):
        if prost(i):
            l.acquire()
            t.value += 1
            l.release()
    print('Proces {} je završio'.format(p.name))
    return
```

Pokrenemo li program u komandnom promptu primijetit ćemo da prvi proces posao završi puno prije posljednjeg procesa. Razlog tomu jest činjenica da prvi proces prebraja proste brojeve među daleko manjim brojevima nego posljednji proces. Primjerice ako imamo 4 procesa i brojimo proste brojeve do 1000000 prvi proces će tražiti proste brojeve među brojevima 2 do 250000 dok će posljednji proces tražiti proste brojeve među brojevima 750000 do 1000000. Brojevi od 2 do 250000 imaju u prosjeku manje djelitelja a i radi se o manjim brojevima pa je u prosjeku potrebno manje vremena da se za jedan broj provjeri je li prost.

Dakle, posao brojanja prostih brojeva mogli bismo dodatno ubrzati kada bismo intervale unutar kojih procesi broje proste brojeve "pravednije" raspodijelili.

Jedan od načina da to napravimo bio bi primjerice da veličine intervala ovise o veličinama brojeva, takva promjena intervala čini se dosta složena. Drugi način bio bi da napravimo više manjih intervala i u tom slučaju više procesa. Međutim, samo kreiranje procesa zahtjeva neko vrijeme i to bi moglo cijeli posao u konačnosti usporiti.

Druga ideja je laka za implementaciju ali bi bila efikasnija kada ne bismo za svaki takav kraći interval morali kreirati poseban proces. Za tu i slične situacije koristit ćemo Pythonovu klasu **Pool**. Osnovna ideja klase **`Pool`** je definirati određeni broj *radnika* koji će odrediti određeni broj *poslova*. Dakle, kreirat ćemo *m* poslova, a za njihovo obavljanje imat ćemo na raspolaganju *n* procesa (radnika). Svaki od procesa uzimat će po jedan posao sve dok se svi poslovi koje smo kreirali ne odrade. Analogija s ovim principom bila bi primjerice da 5 ispravljača treba ispraviti 200 testova. Svaki ispravljač uzme po jedan ispit, kada ga ispravi uzme sljedeći ispit itd. dok svi ispiti ne budu ispravljeni. Ideja je naravno da broj poslova (*m*) bude veći od broja procesa (*n*). U našem slučaju će poslovi biti intervali unutar kojih treba prebrojati proste brojeve. Kreirat ćemo *m* intervala na način da osnovni interval (*a*, *b*) podijelimo na *m* dijelova. A potom ćemo kreirati m procesa koji će redom izvoditi kreirane poslove.

Jedan od parametara klase **`Pool`** je broj procesa koji će izvoditi posao (*n*). Nakon što kreiramo objekt tipa **`Pool`** trebamo definirati:

- što će svaki proces raditi (**`target`** kod klase **`Process`**)
- listu poslova

Nad objektom tipa **`Pool`** definirano je nekoliko metoda koje će nam omogućiti da objektu tipa **`Pool`** dodijelimo što će procesi raditi i listu poslova. Mi ćemo za to koristiti metodu map(). Parametri ove metode su:

- naziv funkcije koju će jedan proces izvoditi
- lista poslova – zadanih kao lista parametara koji će se dodjeljivati funkcijama

Metoda **`map()`** vraća listu vrijednosti. Navedena lista sadržavat će rezultat funkcije za svaki posao. Dakle, u ovom slučaju funkcija koju će proces izvoditi treba vraćati vrijednost. Na kraju će lista koju vraća metoda **`map()`** imati m vrijednosti (za svaki proces po jednu vrijednost).

Isprobajmo ovaj princip za prebrojavanje prostih brojeva. Za početak ćemo izmijeniti metodu **`prosti()`**. Budući da ćemo na kraju imati listu rezultata (po jedan rezultat za svaki posao) nećemo trebati koristiti globalne varijable a funkcija će vraćati broj prostih brojeva na intervalu *a*, *b*. Budući da je svaki posao zadan jednim parom brojeva – jedna varijabla, funkcija **`prosti()`** imat će samo jedan parametar (p) – par brojeva. Unutar funkcije ćemo od para *p* napraviti dvije vrijednosti (a i b) naredbom **`a, b = p`**:

```
def prosti(p):
    a, b = p
    t = 0
    for i in range(a, b):
        if prost(i):
            t += 1
    return t
```

Kao što smo rekli kreirat ćemo **`Pool`** unutar kojeg će biti *n* poslova:

```
p = Pool(n)
```

Poslovi će biti segmenti jednake veličine – dijelovi glavnog segmenta i bit će ih *m*. Ako trebamo prebrojati proste brojeve na intervalu od *a* do *b* i želimo taj posao podijeliti na *m* dijelova, pripadnu listu poslova kreirat ćemo naredbom:

```
[(a + i * d, a + (i + 1) * d)  for i in range(m)]
```

pri čemu je: **`d = (b - a) // n`**

Na kraju će glavni program imati oblik:

```
if __name__ == '__main__':
    a = 2
    b = 1000002
    n = 4
    m = 1000
    d = (b - a) // m
    poc = time()
    p = Pool(n)
    a = p.map(prosti, [(a + i * d, a + (i + 1) * d)  for i in range(m)])
    kraj = time()
    print('{:.5f}'.format(kraj - poc))
    print('Ukupno brojeva: {}'.format(sum(a)))
```

Primijetimo da smo rezultate poslova pohranili u listu *a*. Svaki element liste *a* je po jedan broj – broj prostih brojeva unutar određenog intervala. Na kraju je ukupni rezultat zbroj svih tih brojeva (**`sum(a)`**).

Na kraju ćemo se uvjeriti da je ukupno vrijeme koje je potrebno za brojanje prostih brojeva od 2 do 1000002 uistinu kraće od svih dosadašnjih načina.

Zanimljivo je izmijeniti funkciju **`prosti()`** na način da dodamo ime procesa te "posao" koji funkcija obavlja:

```
def prosti(p):
    pr = current_process()
    a, b = p
    print('Proces: {}; Posao: {}'.format(pr.name, p))
    t = 0
    for i in range(a, b):
        if prost(i):
            t += 1
    return t
```

Izvođenjem ovog programa u komandnom promptu uvjerit ćemo se da čitav posao odrađuju isti procesi, proces kada završi uzima drugi posao itd.

Zadaci za ponavljanje

1. Navedi primjer posla koji će se moći ubrzati ukoliko ga istovremeno obavlja više osoba. Objasni kako bi se odvijanje takvog posla moglo ubrzati.
2. Napiši niz naredbi u Pythonu kojima ćeš provjeriti koliko jezgri (procesora) ima tvoje računalo.
3. Kod brojanja prostih brojeva paralelnim algoritmom u nekim situacijama dolazi do pogrešnog rješenja (iako su algoritmi korektni). Objasni zašto dolazi do pogrešnog rezultata te kako bismo mogli poboljšati rješenje.
4. Zašto je korištenje Poola efikasniji način brojanja prostih brojeva od primjerice načina kod kojega interval unutar kojega brojimo proste brojeve podijelimo na *n* jednakih dijelova (n je broj procesa) te svaki proces broji proste brojeve unutar svog intervala?

Zadaci za vježbu

1. Napiši program u Pythonu koji će pokretati 20 procesa. Svaki od procesa treba ispisati svoje ime te nasumičan prirodan broj do 100.
2. Napiši program koji će mjeriti vrijeme brojanja prostih brojeva u zadanom intervalu i s danim brojem procesa:

Interval/Broj procesa	[2, 10000]	[2, 100000]	[2, 1000000]
1			
2			
[broj jezgri tvog računala]			
2 * [broj jezgri tvog računala]			

3. Za broj ćemo reći da je armstrongov ako je jednak zbroju k-tih potencija svojih znamenaka, pri čemu je *k* broj znamenaka broja. Zadan je interval brojeva od a do b unutar kojega treba prebrojati armstrongove brojeve. Napiši program koji će paralelno brojati armstrongove brojeve na zadanom intervalu te će mjeriti vrijeme i upisivati rezultate u tablicu:

Interval/Broj procesa	[1, 10000]	[1, 100000]	[1, 1000000]
1			
2			
[broj jezgri tvog računala]			
2 * [broj jezgri tvog računala]			

4. Prethodni primjer riješi koristeći klasu Pool s određenom veličinom intervala svakog zadatka te brojem procesa:

Interval		[1, 10000]	[1, 100000]	[1, 1000000]
Veličina pojedinog zadatka	Broj procesa			
100	2			
100	4			
1000	2			
1000	4			

5. Osmisli i implementiraj paralelno rješenje za množenje dviju matrica na 2 procesa.