Raziskava preklopa konteksta med operacijskim sistemom in Go okoljem

Aljaž Brodar

Predmet: Sistemska programska oprema, 2025

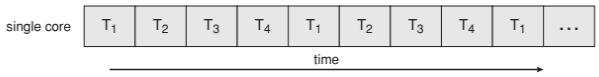
Mentor: Tomaž Dobravec

Kazalo

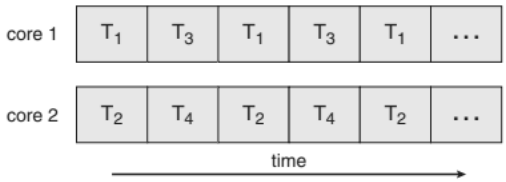
1 Kaj je preklop konteksta?

1.1 Definicija

Prvi računalniki so imeli sposobnost izvajanja le enega programa naenkrat. Ta program je imel popolno kontrolo, nad sistemom in dostop do vseh virov sistema. Na nasprotni strani sodobni računalniki omogočajo, da se več programov naloži v spomin in izvaja sočasno. Ta razvoj je zahteval večjo kontrolo in delitev programov. Te potrebe so razlog za vpeljavo procesov. Proces je program v izvajanju oz. enota dela v modernih sistemih. Sistem je torej sestavljen iz zbirke procesov, kjer se le eden naenkrat izvaja na posameznem procesorju. Na primer procesi operacijskega sistema izvajajo sistemsko kodo in uporabniški procesi izvajajo uporabniško kodo. Ker si želimo, da je procesor čim bolj produktiven, moramo procese, ki se na njem izvajajo, izmenjevati med seboj. To naredimo s tako imenovanim preklopom konteksta. Gre za postopek, ki shrani stanje procesa z namenom, da bo kasneje obnovljen in nadaljeval z izvajanjem v kasnejšem časovnem trenutku. To omogoča več različnim procesom, da si delijo isto CPE, kar je ključno za izvajanje večopravilnih operacijskih sistemov, kjer se več opravil izvaja sočasno v določenem časovnem obdobju s tako učinkovitostjo, da za človeško oko ustvari iluzijo vzporednega izvajanja. O vzporednem izvajanju govorimo, ko izvajamo več procesov v istem časovnem trenutku, medtem ko sočasno izvajanje omogoča več procesom, da napredujejo. Torej je možno imeti sočasnost brez paralelizma. Sledeči sliki prikazujeta ta koncept, kjer je na prvi prikazana sočasnost, na drugi pa vzporednost[1].



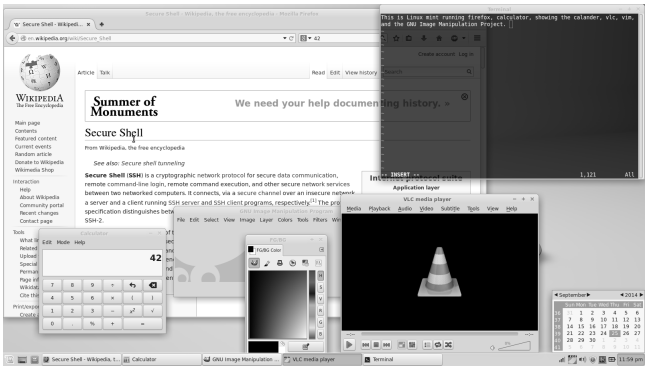
Slika 1: Sočasno izvajanje procesov na enem jedru.



Slika 2: Vzporedno izvajanje procesov na dveh jedrih.

[1] Obenem gre tukaj tudi za sočasno izvajanje na vsakem jedru posebej.

Na ta način ima lahko običajen uporabnik računalnika odprt spletni brskalnik, urejevalnik besedila in podobne programe hkrati in CPE dodeli za izvajanje vsakega nekaj mikrosekund časa, preden preklopi na izvajanje drugega programa, kar na koncu ustvari iluzijo vzporednosti za človeka, ki zazna gibanje v približno 16.7 ms na okvir slike. V tradicionalnem CPE vsak proces uporablja razne CPE registre, da hrani podatke in vzdržuje trenutno stanje izvajajočega se procesa. V večopravilnem operacijskem sistemu le ta zamenjuje med procesi oz. nitmi, z namenom izvajanja več procesov sočasno. Za vsak preklop mora operacijski sistem shraniti stanje trenutnega procesa, čemur sledi nalaganje naslednjega, ki bo tekel na CPE. Temu zaporedju korakov, ki shrani stanje izvajajočega se procesa in naloži novega, pravimo preklop konteksta.



Slika 3: Moderni operacijski sistemi so sposobni obravnave večjega števila različnih procesov ob istem času.

2 Preklop konteksta v OS

2.1 Pomnilniška struktura procesa

Spomin alociran vsakemu procesu je sestavljen iz več delov, ki jim običajno pravimo segmenti. Ti segmenti so sledeči:

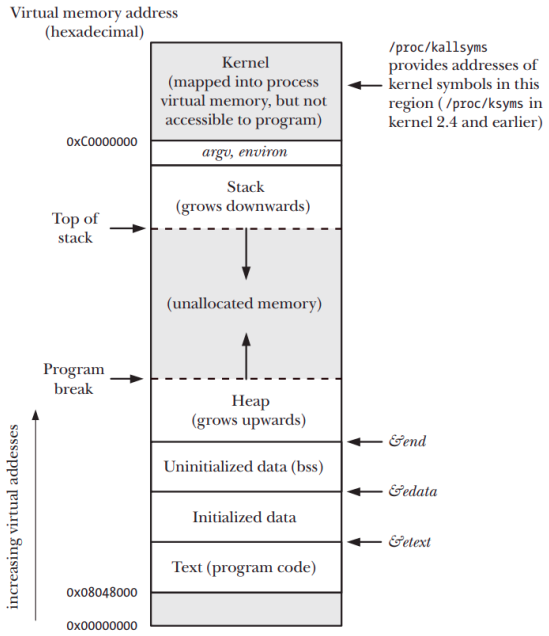
1. **Tekstovni segment**: vsebuje ukaze programa v strojnem jeziku, ki jih proces izvaja. Nastavljen je na *read-only* način, zato da proces nenamenoma ne spremeni svojih ukazov preko nerodne vrednosti kazalca. Ker več procesov lahko izvaja isti program, je ta segment deljen oz. preslikan v virtualni naslovni prostor, več procesov kot ena sama kopija. Iz tega dela je tudi razvidno, da je proces več kot samo programska koda, saj je programska koda tekstovni segment, ki je del procesa. Torej gre za pasivno entiteto napram procesu, ki je aktivna entiteta.

2. **Segment inicializiranih podatkov**: vsebuje globalne in statične spremenljivke, ki so eksplicitno inicializirane. Vrednosti teh spremenljivk so prebrane iz izvršljive datoteke, ko se program naloži v spomin.

3. **Segment neinicializiranih podatkov**: vsebuje globalne in statične spremenljivke, ki niso eksplicitno inicializirane. Preden se program zažene, sistem inicializira cel spomin v tem segmentu na 0. Iz zgodovinskih razlogov se ta segment pogosto imenuje tudi *bss* segment. Ime izhaja iz starejšega zbirniškega mnemonika *"block started by symbol"*. Glavni razlog za ločitev globalnih in statičnih spremenljivk, ki so inicializirane od tistih, ki niso, je, da ko je program shranjen na disku, ni potrebno alocirati prostora za neinicializirane podatke. Namesto tega izvršljiva datoteka samo beleži lokacijo in velikost, potrebno za neinicializirani podatkovni segment, ki se alocira šele, ko program naložimo v izvajalnem času. Na ta način ostane izvršljiva datoteka manjše velikosti, kot bi bila sicer npr. če bi imeli veliko globalno tabelo, ki ni inicializirana nekje v programu.

4. **Sklad**: je segment, ki dinamično spreminja svojo velikost (se širi oz. krči) in vsebuje skladovna okna (angl. *stack frames*). Eno okno se alocira za vsako poklicano funkcijo in hrani njene lokalne spremenljivke t.i. avtomatske spremenljivke, argumente in vrnjeno vrednost.

5. **Kopica**: je del območja kjer lahko v času izvajanja dinamično alociramo spomin. Njen vrh se imenuje *program-break*.

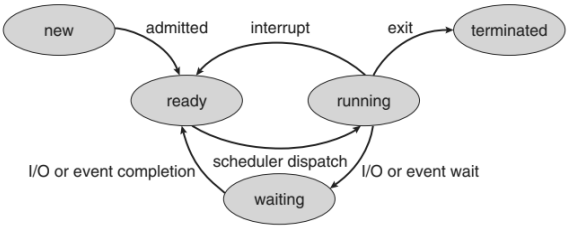


Slika 4: Tipična pomnilniška struktura procesa na Linux/x86-32.

2.2 Stanje procesa

Ko se proces izvaja prehaja med stanji. Stanje procesa je definirano glede na aktivnost, ki jo izvaja. Proces je lahko v enem izmed petih sledečih stanj:

1. **Nov** Proces se ustvarja.
2. **Izvajan** Ukazi se izvajajo.
3. **Čakajoč** Proces čaka na nek dogodek (zaključek V/I ipd.).
4. **Pripravljen** Čaka, da bo dodeljen CPE.
5. **Zaključen** Proces je končal z izvajanjem.



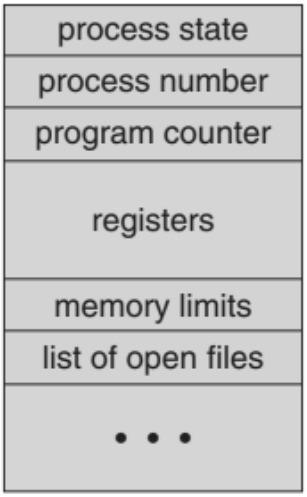
Slika 5: Diagram stanja procesa.

Na vsakem CPE se lahko naenkrat izvaja le en proces, več pa jih je lahko v pripravljenem stanju ali pa na primer v čakajočem stanju.

2.3 Procesni kontrolni blok

Vsak proces se v operacijskem sistemu predstavi s procesnim kontrolnim blokom (angl. *process control block*). Vsebuje mnogo različnih informacij o specifičnem procesu, ki ga predstavlja, vključno s sledečimi:

1. **Stanje procesa** to je lahko nov, izvajan, čakajoč, pripravljen, zaključen.
2. **Programski števec** kaže na naslov naslednjega ukaza, ki bo izveden in je del procesa.
3. **CPE registri** se razlikujejo v številu in tipu. Vključujejo indeksne registre, statusne, splošnonamenske itd..
4. **Informacije za CPE razvrščanje** vsebujejo prioriteto procesa in vse parametre za razvrščanje.
5. **Informacije za upravljanje s pomnilnikom** vsebujejo vrednosti baznih registrov in vse kar je relevantno za delo s pomnilnikom.

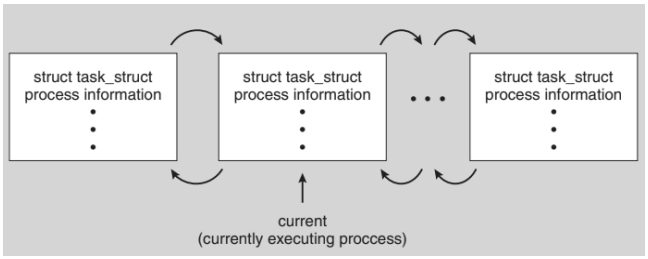


Slika 6: Process control block.

V Linux operacijskem sistemu je PCB predstavljen s C strukturo *task_struct*, ki se nahaja v <*linux/sched.h*>. Nekatera od teh polj so sledeča:

```
long state;                 /* state of the process */
struct sched_entity se;     /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children;  /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;       /* address space of this process */
```

Vsi aktivni procesi v Linuxu so v jedru predstavljeni z dvojnim povezanim seznamom *task_struct* struktur. Jedro vsebuje kazalec *current*. Ta kaže na trenutno izvajan proces v sistemu, kot to prikazuje spodnja slika:

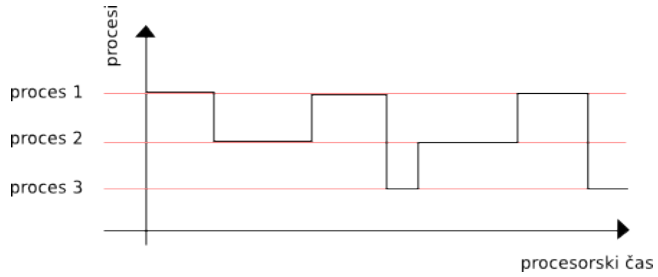


Slika 7: Povezan seznam struktur.

2.4 Koraki pri preklopu konteksta

2.4.1 Kaj sproži preklop konteksta?

1. **Večopravilnost** Razvrščevalni algoritem, ki je del jedra operacijskega sistema, se odloči, kateri proces se bo izvajal v določenem času. Običajno ima sposobnost začasne ustavitve izvajajočega se procesa, tega premakne v vrsto in zažene novega. Tak razvrščevalnik se imenuje razvrščevalnik *brez prekinjanja* (angl. *preemptive*), obstajajo pa tudi t.i. *kooperativni* (angl. non-preemptive) razvrščevalniki, ki procesa v izvajanju nikoli ne prekinejo in zamenjajo z drugim, ampak procesi sami prostovoljno prepustijo procesor drugim procesom. To običajno storijo periodično ali pa ko ugotovijo, da so neaktivni oz. blokirani, ko na primer čakajo na V/I operacijo.



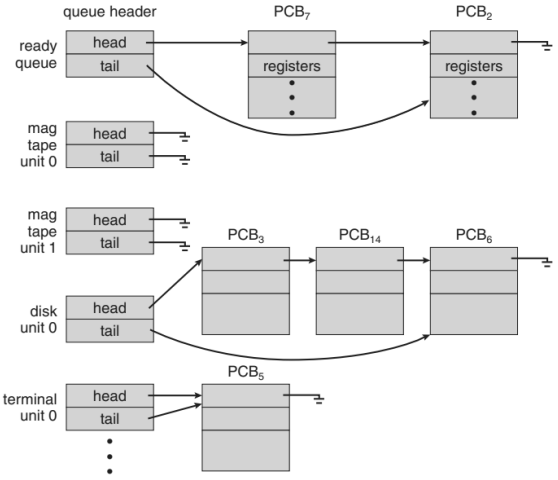
Slika 8: Preklapljanje med tremi procesi, kjer ima drugi razvidno najvišjo prioriteto.

2. **Prekinitve** Ko CPE v sodobnih arhitekturah zahteva podatke z diska, ne potrebuje čakati in periodično preverjati, ali jih je že prejel. Ko bodo podatki pripravljeni, bo namreč prejel prekinitev in bo takrat obdelal svojo zahtevo s pomočjo programa, ki ga imenujemo prekinitveni rokovalnik. Ob prekinitvi strojna oprema avtomatsko zamenja del konteksta, in sicer običajno vsaj toliko, da se lahko sistem vrne v prekinjeni program. V večini primerov se tu menja čim manjši del konteksta z namenom minimizacije skupnega časa, ki je potreben za obdelavo prekinitve. Pogosto jedro za take primere ne ustvari novega, posebnega procesa, ampak se rokovalnik prekinitve izvede le v delnem kontesktu, ustvarjenem ob začetku rokovanja prekinitve.

3. **Menjava med uporabniškim prostorom in jedrom** Ko sistem preklaplja med uporabniškim prostorom in jedrom, menjava konteksta ni potrebna, vendar nekateri operacijski sistemi tudi v takšnih prehodih izvedejo menjavo konteksta.

2.4.2 Razvrščanje procesov med vrstami čakanja

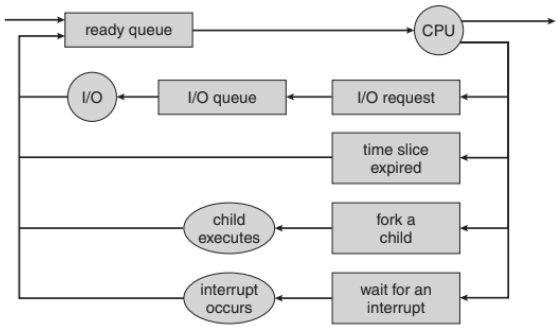
Cilj večprogramiranja je, da je ves čas v izvajanju vsaj en proces, s čimer se doseže večja izraba procesorja. Cilj časovnega deljenja pa je tako pogosto preklapljanje procesorja med procesi, da lahko uporabniki interaktivno sodelujejo z vsakim programom. Da to dvoje dosežemo, mora razvrščevalnik izbrati ustrezen proces za izvajanje na procesorju. Za eno-procesorske sisteme nikoli ne bo več kot en izvajan proces. Če jih je več, bodo preostali morali počakati, da bo procesor prost. Ko se procesi ustvarijo, preidejo v vrsto opravil, (angl. *job queue*), ki vsebuje vse procese v sistemu. Procesi v glavnem pomnilniku, ki so pripravljeni na izvajanje, so na seznamu, ki se imenuje *ready queue*, običajno predstavljen kot povezan seznam. Zaglavje tega seznama vsebuje kazalca na prvi in zadnji PCB v tem seznamu. Sistem obenem vsebuje tudi druge čakalne vrste. Ko je proces alociran, na CPE se izvede za nekaj časa, je prekinjen, konča izvajanje ali pa čaka na dogodek, kot je npr. končana obdelava V/I zahtevka. Recimo, da proces naredi V/I zahtevek za podatke na disku. Ker je na sistemu več procesov, je lahko disk zaposlen s serviranjem enega izmed drugih procesov. Proces bo moral v tem primeru počakati na disk. Seznam procesov, ki čakajo na posamezno vhodno-izhodno napravo, se imenuje *device queue*. Vsaka naprava ima ta seznam.



Slika 9: Ready queue in razni device queue seznami.

Pogosta predstavitev razvrščanja procesov je predstavitev z diagramom vrst. Na spodnji sliki, vrste predstavimo s pravokotniki. V tem primeru imamo dva tipa in sicer ready queue na vrhu in množico device queue seznamo v spodnjih vrsticah diagrama. Krogci predstavljajo vire, ki strežejo posamezni vrsti. In puščice ponazarjajo sam potek pretoka procesov med različnimi vrstami v sistemu. Nov proces se najprej da v ready queue, tam čaka dokler ni izvajan ali

odstranjen. Ko se enkrat izvaja ima na voljo le tri možnosti, kar je enostavno razvidno, če si prikličemo zgoraj omenjeni diagram stanj procesa.



Slika 10: Diagram vrst pri razvrščanju procesov.

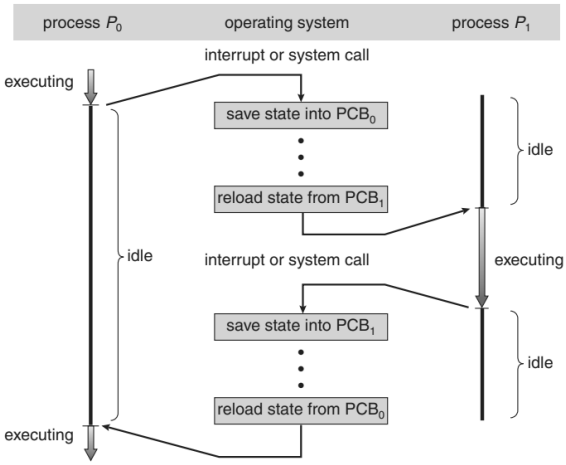
1. Proces lahko ustvari V/I zahtevek in gre v V/I vrsto.
2. Proces lahko ustvari otroški proces in čaka, da se le ta konča.
3. Proces je lahko na silo odstranjen s CPE, kot posledica prekinitve in gre postopoma nazaj v ready queue.

V prvih dveh primerih gre proces v stanje "čakajoč" in potem v "pripravljen", ko hkrati preide v ready queue. Proces ta cikel ponavlja dokler ne zaključi s svojim delom. Ko se to zgodi se ga odstrani z vseh vrst in sprosti se njegov PCB in vse vire povezane z njim.

2.4.3 Izvedba preklopa

Preklop konteksta med procesom P0 in P1 lahko strnemo v sledeče korake:

1. Proces P0 se izvaja in zgodi se prekinitev.
2. Kot odziv operacijski sistem shrani kontekst procesa (PCB) in ustavi izvajanje P0.
3. Operacijski sistem nato naloži PCB drugega procesa P1 in ga začne izvajati na procesorju.
4. Čez nekaj časa se tudi P1 prekine in postopek se ponovi.



Slika 11: Diagram CPE preklopa med procesoma.

Čas namenjen preklopu konteksta je povsem režijski strošek (angl. *overhead*), saj operacijski sistem ne opravlja nobenega neposrednega dela za uporabnika, ko menjuje med procesi. Tipična hitrost je nekaj mikrosekund in je odvisna od strojne opreme sistema.

3 Preklop konteksta v mobilnem okolju

Zaradi omejitev na starejših mobilnih napravah, zgodnje verzije iOS niso omogočale večopravilnosti, kar pomeni, da se v ospredju izvaja le ena aplikacija, vse ostale so prekinjene. Opravila operacijskega sistema pa so se lahko izvajala večopravilno, ker jih je napisal Apple in je lahko zaupal, da se bodo pravilno obnašala. S prihodom iOS 4 leta 2010, Apple omogoča omejeno večopravilnost tudi za uporabniške aplikacije, torej aplikacija, ki je v ospredju in s tem tudi prikazana na zaslonu, se lahko sočasno izvaja z več aplikacijami v ozadju. Apple večopravilnost najverjetneje omejuje zaradi porabe baterije in pomnilnika mobilne naprave. Android teh omejitev nima. Če aplikacija zahteva procesiranje, ko je v ozadju mora uporabiti t.i *service*, ločeno aplikacijsko komponento, ki se izvaja na strani te aplikacije v ozadju. Na primer aplikacija za predvajanje zvoka se prestavi v ozadje in service še vedno nadaljuje s pošiljanjem zvočnih datotek v zvočni gonilnik v imenu te aplikacije. Service nima uporabniškega vmesnika in pusti majhen odtis na pomnilnik, kar omogoča učinkovito večopravilnost.

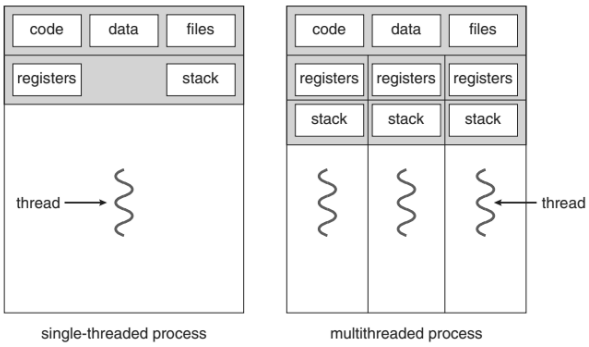
4 Preklop konteksta v GO

4.1 Niti

Program je le niz strojnih ukazov, ki jih je treba izvajati zaporedno enega za drugim. Da bi to omogočil, operacijski sistem uporablja koncept niti. Naloga niti je, da upošteva in zaporedno izvaja niz ukazov, ki so ji dodeljeni. Izvajanje se nadaljuje, dokler ni več ukazov, ki bi jih nit lahko izvedla. Zato nit imenujemo "pot izvajanja".

Vsak program, ki ga zaženete, ustvari proces, vsak proces pa dobi začetno nit. Niti imajo sposobnost ustvarjanja novih niti. Vse te različne niti tečejo neodvisno druga od druge odločitve, o razporeditvi pa se sprejemajo na ravni, niti ne na ravni procesa. Niti lahko tečejo sočasno (vsaka po vrsti na posameznem jedru) ali vzporedno (vsaka teče istočasno na različnih jedrih). Niti tudi ohranjajo svoje stanje, da omogočajo varno, lokalno in neodvisno izvajanje svojih navodil.

Nit sestavlja njen ID programski števec, množica registrov in pa sklad. Z drugimi nitmi, ki pripadajo istemu procesu si deli kodno sekcijo (angl. *code section*), podatkovno sekcijo (angl. *data section*) in ostale vire operacijskega sistema, kot so odprte datoteke in signali. Tradicionalni proces ima na začetku le eno nit. Če jih kasneje ustvari več, lahko opravlja več kot eno opravilo naenkrati. Spodnja slika prikazuje razliko med eno nitnim in večnitnim procesom.



Slika 12: Primerjava procesa z eno nitjo s procesom, ki ima več niti.

Niti so lahko v treh stanjih:

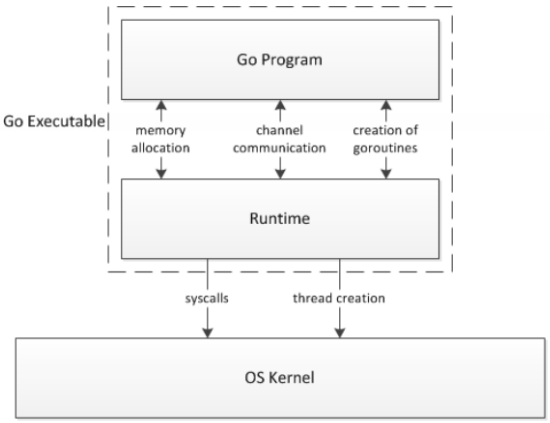
1. Nit se izvaja na CPE.
2. Nit je izvedljiva.
3. Nit čaka (na nek dogodek, npr. V/I).

Hkrati je pri obravnavi preklopa konteksta pomembno razlikovati med *CPU-Bound* in *IO-Bound* nitmi. Pri prvih se nikoli ne zgodi, da bi prišle do točke, kjer morajo čakati na odgovor na neko zahtevo, torej so vedno v stanju izvajanja. Edina omejitev na njihovo hitrost izvajanja je torej zmogljivost CPE. Medtem ko se pri slednji vrsti niti lahko zgodi, da so začasno ustavljene, recimo zaradi V/I zahtevka, in ne morejo nadaljevati, dokler se ta ne obdela. Torej se jih prestavi v čakajoče stanje in posledično se zgodi preklop konteksta, ki jih odstrani z izvajalne enote.

4.2 Izvajalno okolje GO

GO programi so prevedeni v strojno kodo preko GO prevajalniške infrastrukture. Ker GO podpira visokonivojske strukture, kot so gorutine, kanali in čiščenje spomina, (angl. garbage collection) potrebuje dodatno programsko opremo t. i. GO runtime infrastrukturo, da jih podpre. To pomeni, da ko napišemo GO program se ne izvede povsem samostojno, ampak potrebuje dodatno kodo, napisano v tem primeru v C. V fazi povezovanja se ta C izvajalna koda združi direktno z našim programom in rezultat je ena sama izvedljiva datoteka, ki vsebuje vse kar potrebuje, da se izvede. Ko izvedemo GO program, operacijski sistem vidi le samostojen program, ne pa nekaj, kar bi bilo odvisno od zunanjih knjižnic.

GO program v izvajanju si lahko predstavljamo kot konstrukt, sestavljen iz dveh plasti: uporabniške kode in izvajalnega okolja (angl. runtime) ki komunicirata preko funkcijskih klicev za upravljanje gorutin kanalov in drugih višjenivojskih abstrakcij.



Slika 13: Struktura GO programa in njegovega izvajalnega okolja.
Vir: [1]

Eden pomembnejših delov GO runtime je gorutine scheduler. Izvajalno okolje beleži vsako gorutino in jih razvršča na niti, ki pripadajo procesu za izvajanje. Gorutine se razlikujejo od niti in so od njih odvisne, da se sploh lahko izvedejo. Za visoko učinkovitost programa je ključno izvesti ravno to razvrščevanje pravilno. Ideja Gorutin je, da so sposobne sočasnega izvajanja kot niti, vendar so veliko "lažje" v primerjavi. Več niti lahko izvajamo vzporedno, do meje, ki jo določi programer v spremenljivki GOMAXPROCS.

Pomembno je, da vse, kar operacijski sistem vidi, je proces, ki zahteva in izvaja več niti, medtem ko je koncept razvrščanja gorutin na te niti le konstrukt virtualnega okolja v runtime.

V Go runtime so prisotne tri glavne C strukture, ki pomagajo runtime in razvrščevalniku: 1) G struktura - predstavlja eno samo GO rutino. Vsebuje polja za vzdrževanje njenega sklada in statusa ter reference na kodo za izvajanje katere je odgovorna.

```
struct G {
 byte* stackguard; // stack guard information
 byte* stackbase;  // base of stack
 byte* stack0;     // current stack pointer
 byte* entry;      // initial function
 void* param;      // passed parameter on wakeup
 int16 status;     // status
 int32 goid;       // unique id
 M* lockedm;       // used for locking M's and G's
 ...
};
```

2) M struktura - predstavlja nit operacijskega sistema, kot jo vidi Go runtime. Med drugim ima polja, kot je kazalec na globalno vrsto Gorutin, kazalec na trenutno izvajane gorutine ter svoj predpomnilnik.

```
struct M
{
 G* curg; // current running goroutine
 int32 id; // unique id
 int32 locks ; // locks held by this M
 MCache *mcache; // cache for this thread
 G* lockedg; // used for locking M's and G's
 uintptr createstack [32]; // Stack that created this thread
 M* nextwaitm; // next M waiting for lock
 ...
};
```

3) SCHED struktura - gre za globalno strukturo, ki vzdržuje različne vrste G in M struktur in dodatne informacije, ki jih razvrščevalnik potrebuje za pravilno delovanje. Vsebuje dve vrsti G struktur ena je izvedljiva vrsta, kjer niti lahko najdejo delo, druga pa prosta vrsta G struktur. Imamo le eno vrsto za M strukture, ki jih razvrščevalnik vzdržuje niti v tej vrsti so neaktivne in čakajo na delo. Da vse te vrste spreminjamo, mora biti Sched lock polje aktivno.

```
struct Sched {
 Lock; // global sched lock .
 // must be held to edit G or M queues
 G *gfree; // available g' s ( status == Gdead)
 G *ghead; // g' s waiting to run queue
 G *gtail; // tail of g' s waiting to run queue
 int32 gwait; // number of g's waiting to run
 int32 gcount; // number of g's that are alive
 int32 grunning; // number of g's running on cpu
 // or in syscall
 M *mhead; // m's waiting for work
 int32 mwait; // number of m's waiting for work
 int32 mcount; // number of m's that have been created
 ...
};
```

Runtime se začne z več Gorutinami. Ena skrbi za čiščenje spomina (angl. garbage collection), druga razvrščevanje in tretja predstavlja uporabnikovo Go kodo. Hkrati je na začetku dodeljena nit M. Ko program teče, se lahko ustvarijo dodatne gorutine preko uporabnikovega Go programa in več niti je morda potrebnih, da izvedemo vse. Na primer, da koda zahteva, da nit blokira, to se lahko zgodi recimo če pokličemo sistemski klic, potem se nova nit vzame iz

vrste neaktivnih niti. To se naredi zato, da zagotovimo, da vse gorutine, ki so še vedno izvedljive, nimajo blokiranega izvajanja zaradi pomanjkanja niti.

4.3 GO razvrščevalnik

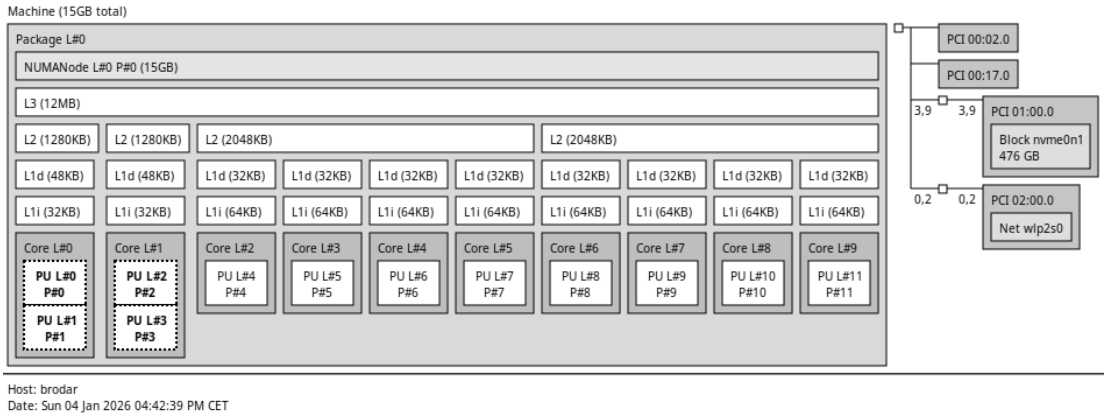
Ko zaženemo GO program, se za vsako jedro gostiteljevega sistema, dodeli t.i. logično jedro (označimo P) [2]. Koliko takšnih jeder imamo lahko na Linux operacijskem sistemu preverimo z ukazoma:

```
$lscpu
$lstopo
```

```
Architecture:             x86_64
CPU op-mode(s):           32-bit, 64-bit
Address sizes:            39 bits physical, 48 bits virtual
Byte Order:               Little Endian
CPU(s):                   12
On-line CPU(s) list:      0-11
Vendor ID:                GenuineIntel
Model name:               13th Gen Intel(R) Core(TM) i7-1355U
CPU family:               6
Model:                    186
Thread(s) per core:       2
Core(s) per socket:       10
Socket(s):                1
Stepping:                 3
CPU(s) scaling MHz:       36%
CPU max MHz:              5000,0000
CPU min MHz:              400,0000
```

Slika 14: Izpis ukaza lscpu.

[2] Ob prisotnosti procesorja z več fizičnimi nitmi na jedro (angl. *hyper-threading*), bo vsaka nit Go programu predstavljena kot virtualno jedro.



Slika 15: Izpis ukaza lstopo.

Opazimo, da ima obravnavan sistem eno CPE, ki ima deset jeder, kjer imajo vse razen prvih dveh po eno nit [3]. Prvi dve jedri pa imata vsaka po dve niti, kar tudi pomeni da ima procesor možnost *Hyper-Threading* [4]. GO program torej vidi 12 logičnih jeder (P), kar pomeni da se lahko vzporedno izvaja največ 12 niti operacijskega sistema, kjer se vsaka OS nit preslika na fizično nit. To lahko v GO preverimo z ukazom *runtime.NumCPU()*.

Vsakemu logičnemu jedru (P) se dodeli OS nit (označimo M [5]). To nit upravlja operacijski sistem, ki je med drugim odgovoren za njeno preslikavo na fizično jedro, kjer se bodo njeni ukazi izvajali.

Hkrati se vsakemu GO programu dodeli začetno GO rutino (označimo G), ki je enota izvajanja GO programa. gorutine si lahko predstavljamo kot niti na aplikacijskem nivoju, ki so po arhitekturi zelo podobne OS nitim. Tako kot se za OS niti izvede preklop konteksta na jedru, se za gorutine izvede preklop konteksta z OS niti (M).

G gredo v vrsto za izvajanje oz. *run queue*. GO razvrščevalnik ločuje dva tipa teh vrst in sicer *Global Run Queue* (GRQ) in *Local Run Queue* (LRQ). Vsakemu P se dodeli LRQ v katerem so G, ki so dodeljene v to vrsto z namenom, da bodo na tem P tudi izvedene. Te G se s preklopom konteksta izmenjuje z M, ki je takrat na P. GRQ vsebuje vse G, ki še niso bile dodeljene nobenemu P. V GO obstaja proces, ki G iz GRQ premika v ustrezne LRQ.

Razvrščevalnike delimo na več vrst. Za potrebe tega poglavja obravnavamo dve vrsti in sicer razvrščevalnik s prekinjanjem (angl. *preemptive scheduler*) ter sodelovalni razvrščevalnik (angl. *cooperative scheduler*). Tipičen primer razvrščevalnika s prekinjanjem je razvrščevalnik operacijskega sistema. Ta ima možnost kadarkoli zamenjati trenutno nit na CPE z drugo. Z drugimi besedami

[3] Tu govorimo o fizičnih nitih procesorja, ne o nitih operacijskega sistema.

[4] Vsaka nit operacijskega sistema ima lasten sklad (stack) v pomnilniku. Fizične niti procesorja (hardware threads) nimajo lastnega sklada, temveč zgolj lasten izvajalni kontekst; pri izvajanju uporabljajo sklad trenutno dodeljene niti operacijskega sistema.

[5] M kot *machine*.

niti se lahko na silo zaustavi, čeprav bi lahko te glede na vsebino programa lahko nadaljevale s svojim izvajanjem. Običajno se preklop konteksta izvede ob poteku časovne rezine ali pa ko ima neka nit višjo prioriteto. Če razvrščevalnik sposobnosti da ob poljubnem času zahteva preklop ne bi imel, kot jo na primer nima pri sodelovalnem razvrščanju [6], potem tvegamo stanje, kjer si med drugim ena nit lahko povsem prisvoji CPE. Sodelovalni razvrščevalnik, dopušča niti več svobode, saj ta lahko teče dokler same ne preda nadzora nad CPE ali dokler ne začne blokirati. Oba modela imata svoje prednosti in slabosti. Model s prekinjanjem deluje bolje, ko imamo kopico programske opreme, ki med seboj ni povezana, model s sodelovanjem pa, ko se izvajajo programi, ki so bili načrtovani tako, da med seboj sodelujejo.

GO razvrščevalnik je del GO Runtime [7], le ta je vgrajen v aplikacijo. To pomeni, da se izvaja v uporabniškem prostoru ("nad jedrom"). Pri njem gre za sodelovalni model razvrščanja, vendar kljub temu ima posebnost, da se navidez obnaša kot model s prekinjanjem. Razvijalec ne more predvideti, kaj bo GO razvrščevalnik naredil, ker to odločitev opravi GO runtime. Programsko opremo mora posledično razvijati, tako kot da gre za razvrščanje s prekinjanjem. Razvrščanje se zgodi le ob natančno definiranih "varnih" točkah.

Pri običajnih programskih jezikih, s katerimi rešujemo probleme sočasnosti, bi se večnitenje zgodilo tako, da bi izvajalno okolje (*runtime*), razdelilo opravila na niti in prepustilo vso logiko upravljanja z njimi operacijskem sistemu. Ta bi se torej odločil na kakšen način se bodo izvedle. Če so niti *IO-Bound*, bi razvrščevalnikovi preklopi konteksta omogočili, da se te niti izvajajo skoraj oz. navidezno paralelno in potemtakem dobimo velik doprinos k učinkovitosti v primerjavi z naivnim pristopom, kjer bi jedro postalo neaktivno vsakič, ko nit na njem čaka na V/I zahtevek. Po drugi strani, če ima proces več niti, ki si delijo isto jedro in so vse te niti *CPU-Bound*, bi rezultat preklopov konteksta bil drastično slabša učinkovitost procesa. Zato izvajanje veliko izračunov, ki vezani na CPE sočasno običajno ni dobra ideja.

Glavni kompromis oz. razmerje, ki se nadzira na aplikacijskem nivoju je med režijskimi stroški preklopa konteksta ter časom, ko je program razvijalca neaktiven. Preklop konteksta je najbolj učinkovit, ko imamo opravka predvsem z *IO-Bound* nitmi. Te naredijo režijske stroške preklopa konteksta neizogibne na nivoju operacijskega sistema. Glavno vprašanje postane, kaj če bi lahko te stroške omejili na aplikacijskem nivoju? Izvajalno okolje ima več nadzora nad nitmi aplikacije, ki ga GO-jev runtime razvrščevalnik v svoj prid tudi uporabi. Z naprednimi tehnikami [8], doseže, da *IO-Bound* gorutine razvrščevalniku operacijskega sistema nit na kateri se izvajajo, predstavijo navidez kot *CPE-Bound*. Z vidika OS so njegove niti vedno zaposlene z delom, čeprav je to delo sestavljeno iz dela večih gorutin. Ker je preklop konteksta v GO veliko bolj poceni kot pa v OS, je GO runtime uspelo zmanjšati režijske stroške preklopa konteksta in hkrati

[6] Sodelovalno razvrščanje spada v večjo družino razvrščanja imenovano razvrščanje brez prekinjanja (angl. *non-preemptive scheduling*).

[7] Izvajalno okolje (angl. *runtime*) je programska oprema zasnovana za podporo pri izvajanju računalniškega programa zapisanega v enem izmed programskih jezikov.

[8] *networking pooling*, *work stealing*, ...

zagotoviti zelo majhen čas, ko je program neaktiven na jedru CPE. Posledično, GO pridobi na učinkovitosti na zgoraj omenjenem kompromisu. Z drugimi besedami Go runtime prestavi večino razvrščanja sočasnega IO-bound dela v uporabniški prostor, kjer so preklopi med gorutinami cenejši, s čimer zmanjša število dragih preklopov na ravni operacijskega sistema in izboljša učinkovitost pri visoki sočasnosti.

5 Analiza preklopa konteksta niti v GO

Eksperimentalno si lahko pogledamo trajanje konteksta v GO s preprostim programom, kjer imamo dve gorutini, ki si med sabo preko skupnega kanala [9], pošiljata gorutine. Ker je kanal v podani programski kodi, ustvarjen s prazno velikostjo to pomeni, da blokira branje dokler druga gorutina vrednosti ne pošlje v kanal. Obenem blokira pisanje, dokler na drugi strani prejemnik ni pripravljen na branje. Branje se v GO označi z levo usmerjeno puščico na levi strani kanala *<-ch*, pisanje pa z enako usmerjeno puščico na desni strani kanala in vrednostjo, ki jo v kanal želimo zapisati *ch<-val*.

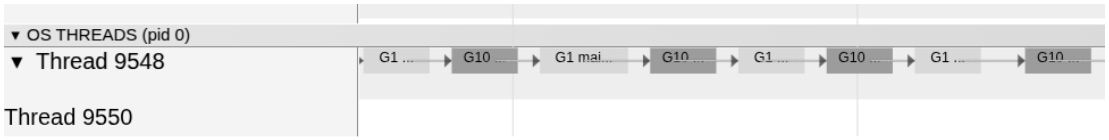
[9] Kanali v GO so analogni cevovodu, ki povezuje sočasne gorutine.

```
package main

import (
"fmt"
"os"
"runtime"
"runtime/trace"
"time"
debug "runtime/debug"
)

func child(c chan string) {
        for msg := range c {
                c <- msg
        }
}

func main() {
        f, err := os.Create("trace.out")
        if err != nil {
                panic(err)
        }
        defer f.Close()

        if err := trace.Start(f); err != nil {
                panic(err)
        }
        defer trace.Stop()

        runtime.GOMAXPROCS(1)
        debug.SetGCPercent(-1)

        c := make(chan string)
        go child(c)

        const niters = 2000000
        for i := 0; i < niters; i++ {
                c <- "test"
                reply := <-c
                if len(reply) != 4 {
                        panic("err")
                }
        }

        fmt.Println("done")
        time.Sleep(1 * time.Second)
}
```

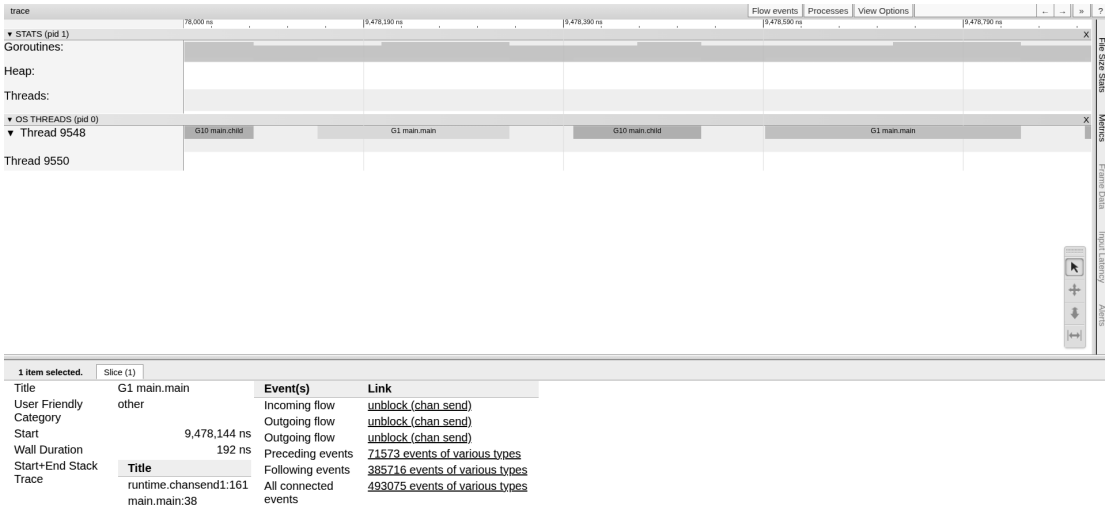
Listing 1: Go example code

V programu imamo dve gorutini in sicer začetno *main.main* ter *main.child*, kjer slednjo sami ustvarimo z *go child(c)*, medtem, ko je prva avtomatsko ustvarjena kot začetna nit procesa. Na začetku nastavimo, da se vzporedno lahko izvaja le ena niti. Nato ustvarimo *main.child*, ki periodično bere iz kanala v zanki, dokler se le ta ne zapre. Kot zgoraj omenjeno, kanal ob določenih bralnih oz. pisalnih situacijah blokira in takrat se izvede preklop gorutin na M. S

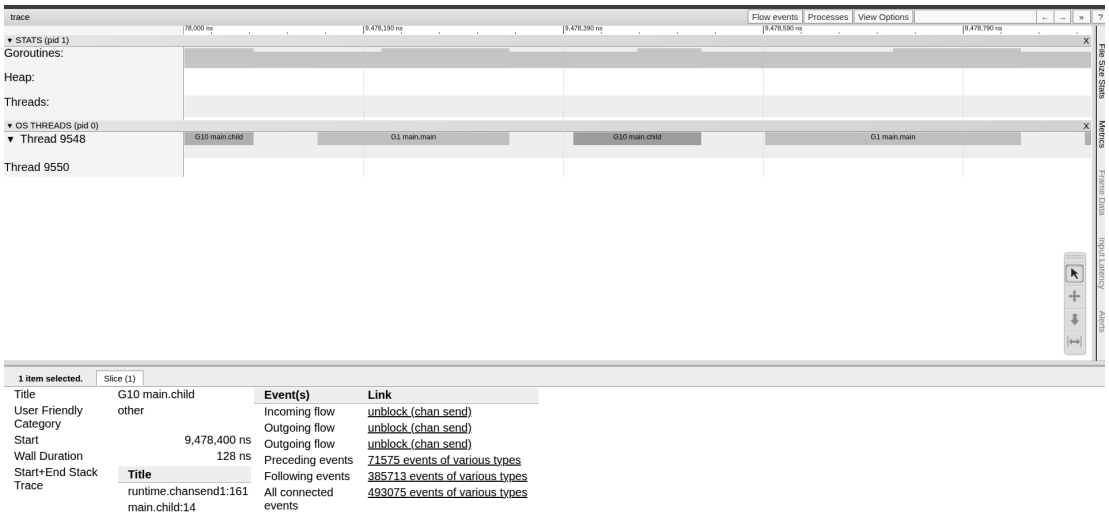
pomočjo orodja *trace* si lahko to natančneje pogledamo.



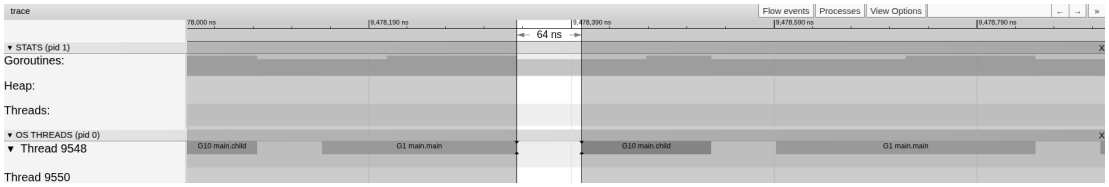
Slika 16: Menjava gorutin main.main in main.child na isti niti.



Slika 17: Trajanje gorutine main.main.



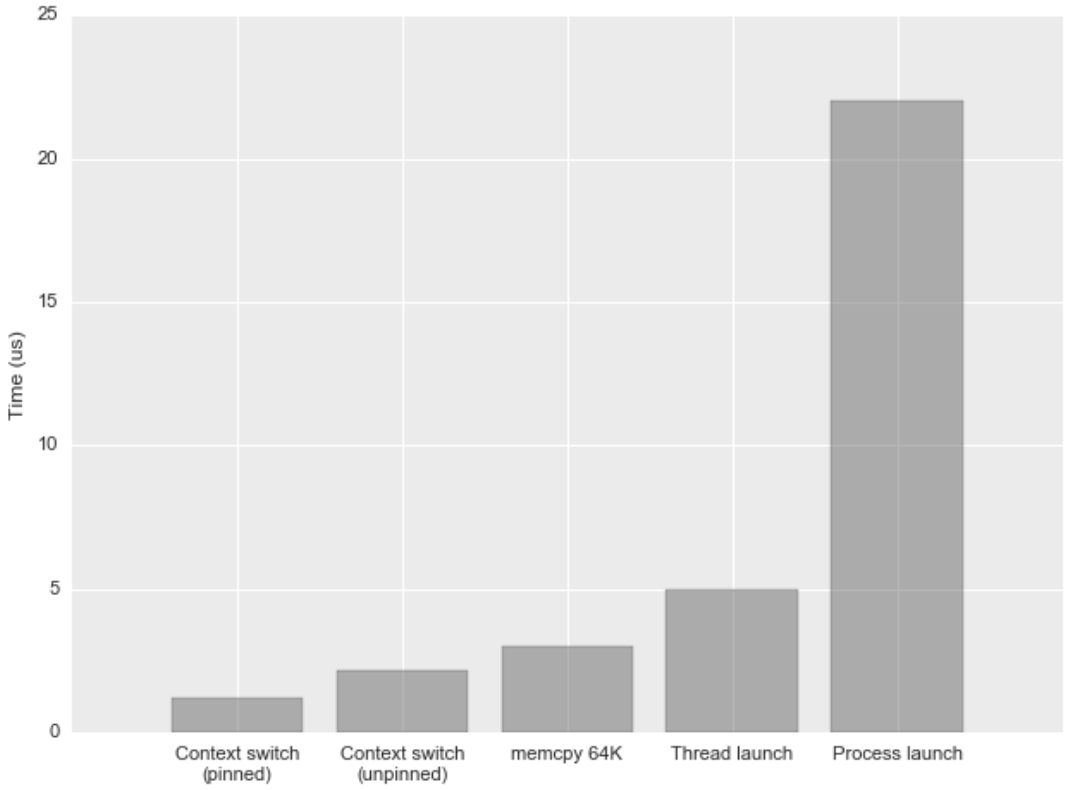
Slika 18: Trajanje gorutine main.child.



Slika 19: Trajanje preklopa med main.main in main.child.

Opazimo da se delo *main.main* izvaja na niti 9548 približno 192 ns, medtem ko *main.child* okrog 128 ns. Čas med tema dogodkoma pa se porabi za menjavo teh gorutin in traja približno 64 ns. Za primerjavo z Linux OS, z enakim programom v C, kjer pride do preklopa niti na nivoju OS, dobimo veliko manjšo prepustnost, preklop tam traja okoli 1.5 μs[10], kar je skoraj 24-krat več. Torej za podano strukturo programa je preklop konteksta niti OS približno 24-krat dražji, kot trajanje preklopa gorutin na OS niti. Za primerjavo, kopiranje 64 KiB na istem računalniku traja trikrat več kot omenjen OS preklop konteksta.

[10] Glej (Bronshtein, 2018).



Slika 20: Trajanje preklopa med main.main in main.child.

6 Literatura

1. Deshpande, N., Sponsler, E., & Weiss, N. (2011). *Analysis of the Go runtime scheduler*. Columbia University, str. 1–3. Dostopno na: `https://www.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf`

2. Bronshtein, E. (2018). *Measuring context switching and memory overheads for Linux threads*. Dostopno na: `https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/`

3. Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco: No Starch Press.

4. Silberschatz, A., Galvin, P. B., & Gagne, G. (1994). *Operating System Concepts* (2. izdaja). Reading, MA: Addison-Wesley.