

*Prolećni semestar, 2016/17*

*PREDMET: CS225 OPERATIVNI SISTEMI*

*Projektni zadatak*

Implementacija i analiza konkurentnosti u Python-u

Ime i prezime: **Nemanja Kuzmanovic**

Broj indeksa: **2851**

Profesor: **Doc. dr Selena Vasić**

Beograd 2017.

# Abstrakt:

Kako bi se lakše objasnio, i razumeo problem niti (threadova) u višenitnim (multithread) programima, često se koriste nekoliko postavljenih problema. Između ostalog i ’’readers-writers problem’’.

U ovom projektu, detaljnije će se analizirati i objasniti sam problem, šta je i šta predstavlja, i kako ga možemo efektivno rešiti. Dakle, najefektivniji način rešavanja problema je upotreba semafora.

Projekat će sadržati detaljniju analizu i objašnjenje semafora kako bi se i problem i semafori bolje razumeli, posle čega će biti izvršena implementacija samih semafora tako što će se rešiti gore pomenuti već postavljeni problem. Implementacija će biti izvršena u programskom jeziku python, iz nekoliko razloga,prvenstveno zato što već ima dosta izvora za C jezik kao i Javu, pa će ovo biti odlična prilika da se implementacijom u drugi jezik semafori bolje i dodatno savladaju. Takođe, rešavanje ovog problema sam izabrao zato što se putem njega detaljno može ova oblast obraditi bez da se odlazi previše u abstrakciju implementacijom nekih većih problema, a takođe pokriva suštinu.

Dopuna:

U razgovoru i dogovoru sa asistenom, data mi je sugestija tj. Predlog da obradim još neki deo pored semafora ne samo same semafore, kao npr Lockove, te sam došao na ideju da obradim čitavu oblast koja se vezuje za konkurentne procese, analizom postojećeg koda nad koji sam implementirao svoj kod koji je pisan za potrebe drugog projekat, te je ovde iskorišćen kako bi se izvršila analiza.

# Uvod:

Dakle, kao što je u dopuni abstrakta navedeno, projekat ipak neće biti samo okrenut ka semaforima nego će biti obrađeni još neki konkurentni procesi kroz python skripting jezik. Kod je uzet iz nekoliko raličitih izvora, pored interneta, koristio sam i deo svog projekta za potrebe drugog predmeta. Ideja je bila da se tako implementiran kod analizira detaljno kako bi se konkurentni procesi bolje savladali. Nije se implementirala sama mehanika tj. Algoritam ovih proces jer bi to iziskivalo ogromno znanje prvenstveno a potom i iskustvo, vreme kao i veliki tim iza sebe, te je odlučeno da se kroz manji program impementiraju ovi procesi te analizom pokažu da je student savladao deo gradiva. Još jedna napomena za kraj uvoda, pre nego što krenemo sa samom implementacijom, je to da sam uglavnom nalazio kod za C ili C++ jezik kao i za Javu, što je bila još jedna motivacija i odlična podloga da se krene u implementaciju u Python-u koji iz godine u godine grabi sve više, brže, i bolje ka vrhu top liste jezika, s razlogom.

# Implementacija:

Kako je projekat praktičnog tipa, neću koristiti teoriju da pokrijem ono što radim, smatram da je to više nego dobro i odlično objašnjeno u samim predavanjima vezanim za predmet, te bih za takvu teoriju preporučio predavanja iz ovog predmeta, a u suporotnom ako bih se bavio tom teorijom i pisao o tome, citirao bih veliki deo iz samih predavanja.

Python – sinhronizacija niti: Locks, Rlocks, Semaphores, Conditions, Events, Queues

Dakle, u ovom projektu, bavićemo se mehanizmima za sinhronizaciju niti. Tipovi koju su gore navedeni, će biti tipovi koji će se obraditi, te će se objasniti što detaljnije značenje iza samih mehanizama.

Za sam početak kako bi se krenulo od nečeg, biće predstavljen jednostavan program koji kroz nekoliko niti, ulazi na sajtove i kupi sadržaj sa njih. Kod i program su dosta uprošćeni od onoga šta se koristilo u drugom projektu jer su te ostale stvari irelevantne za ovaj predmet. Problem koji je nastao tokom skrepovanja (skidanje sadržaja sa određenog sajta) sajta, i upisivanja u bazu podataka i fajl, je bio taj, da su se implementirne niti prekidale, tačnije prelamale su u određenim trenutcima jedna drugu. Program koji sam radio je skidao dakle sadržaj sa sajta kroz nit i upisivao u bazu, međutim pošto se sinhrono skidalo i upisivalo u bazu, a isto tako se u toku skidanja ukoliko se otvori određen segment i čitao sadržaj koji se odmah ispisuje na sajt a ne u bazu, dolazilo je do čudnih stvari gde bi kod pukao zbog niti ili bi izbacivao ono šta je traženo na potpuno pogrešan način, koji sam eventualno uspeo da rešim i biće dalje prikazan u nastavku. Problem je, pretpostaviću, bio u tome što sam skidao preko 100 000 podataka u sekudni kroz nekoliko ne kontrolisanih threadova... Za početak jednostavan program koji pokazuje rad sa nitima.

Imamo program koji skida dakle sadržaj sa određenog sajta i upisuje ga u određeni fajl. Ovo se može odraditi i bez niti naravno, međutim kako bi se dosta ubrzao ceo proces, pogotovo kada je veliki broj informacij u pitanju, koriste se niti. Kreiraćemo dve niti koje će obraditi ove procese skidanja.

Klasa FetchUrls je bazirana na niti i prihvata lisu URL-a ka sajtovima čiji sadržaj želimo da skinemo, te fajl u koji ćemo taj sadržaj i ispisati.

U konkretnom programu koji sam radio su kupljene konkretne informacije vezane za određene proizvode, a kako nisam u mogućnosti da prikažem kroz ovaj kod sve to skidanje, koje je irelevantno, uzeću nekoliko drugih sajtova kroz koje ću ukratko pokazati ove mehanizme, te ću uraditi ono što je najjednostavnije i praktičnije a to je skinuti sadržaj celog sajta umesto konkretnih segmenata koji su korišćeni u pomenutom projektu.



Slika 1: Niti

U main metodi možemo videti da se pokreću dve niti koje će skidati ovaj sadržaj.



Slika 2: Output

Problem koji nastaje ovakvim skidanjem sadržaja je sličan na koji sam i ja naleteo u mom projektu, a to je da će niti pored skidanja u isto vreme upisivati u fajl, te će nastati potpuni haos, što naravno ne želimo!

Da bi ovakav problem rešili korisili smo ’’brave’’ (eng. Lock) kako bi zaključali jedni nit dok upisuje u fajl, a potom otključali, te zaključali drugu i omogućili joj da upise sadržaj. Ovim putem razdvajamo taj nastali haos kada dve niti istovremeno upisuju u fajl. Brava ima dva stanja, zaključano i otključano, a metode koje se mogu koristiti za manipulaciju njima su: aquire() i release(). U teoriji, postoje nekoliko situacija, šta se dešava kada koristimo ove metode:

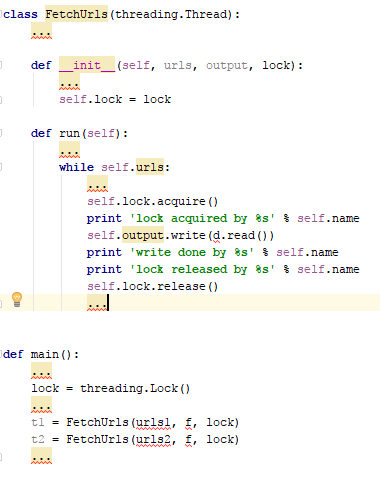
* Ukoliko je stanje otključano: aquire() menja stanje u zaključano.
* Ukoliko je stanje zaključano: poziv aquire() blokira dok druga nit ne pozove release()
* Ukoliko je stanje otključano: poziv release() nam vraća izuzerak (RuntimeError exception).
* Ukoliko je stnje zaključano: poziv release() menja stanje na otključano.

Dakle, postoje određena stanja posle kojih moramo određenu metodu pozvati kako ne bi došlo do ne predviđenih situacija i akcija.

Vratimo se na naš problem. Kako bi rešili problem u kom dve niti upisuju sadržaj istovremeno u isti fajl, moramo proslediti argument lock u konstruktor naše FetchUrls klase, koji koristimo kako bi zaštitili operaciju upisa u fajl.

Implementiramo:

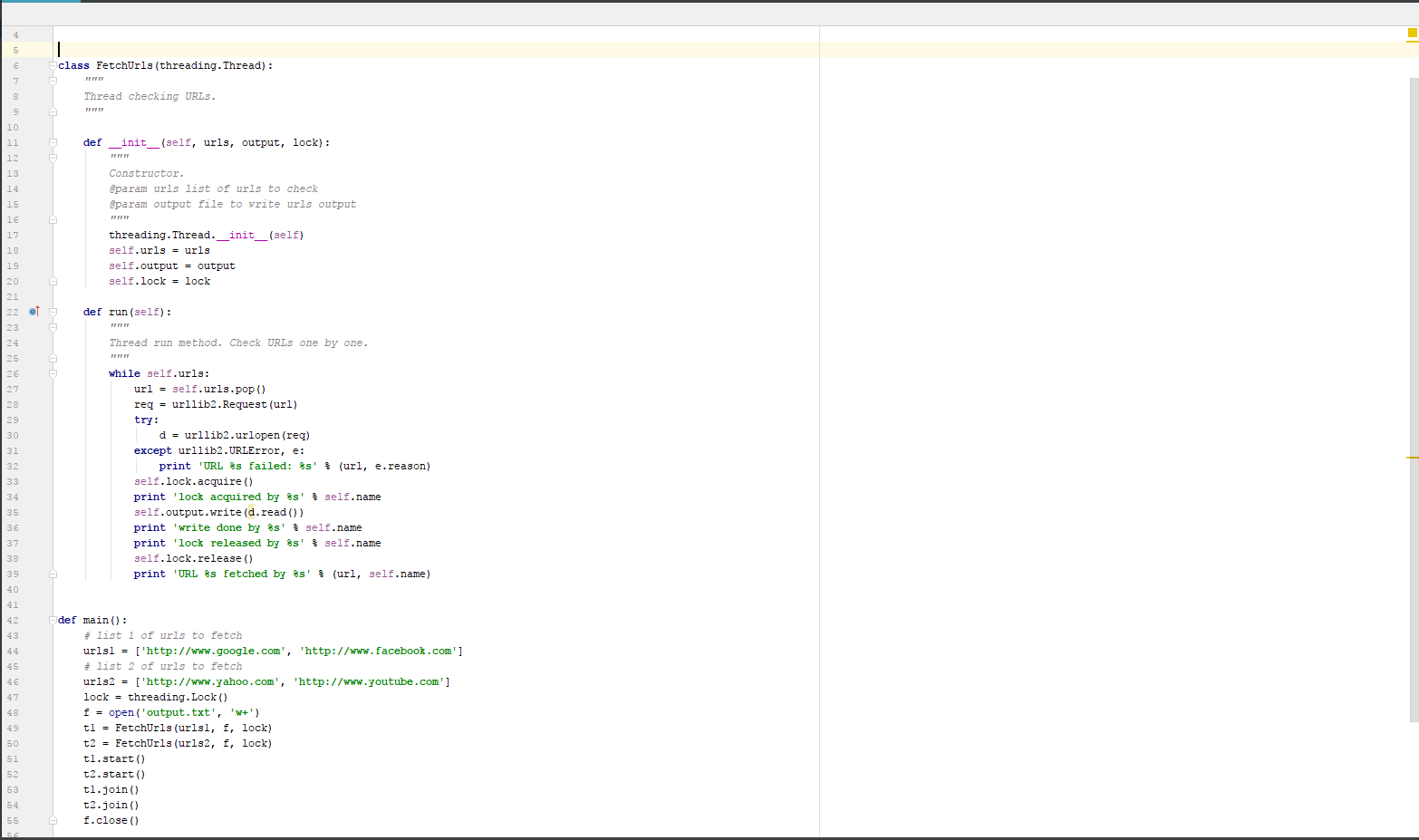
\*Tačkice predstavljaju prethodno implementiran kod koji nećemo u ovom delu ponoviti, te samo prikazujemo novo imlementirani.



Slika 3: Implementacija lock-a

Na slici iznad se mogu videti samo delovi koji se implementirali kako bi zaključali te otključali potrebnu nit.

Slika ispod prestavlja objedinjene, dakle, prethodno implementirane niti sa sadašnjim implementiranim Lockom koji nam omogućuje da bezbedno sa dve niti prvo pokupimo sadržaj sa sajta te ga upišemo nit po nit u sam fajl, kako istovremeno ne bi napravili haos u samom fajlu.



Slika 4: Threads and Lock

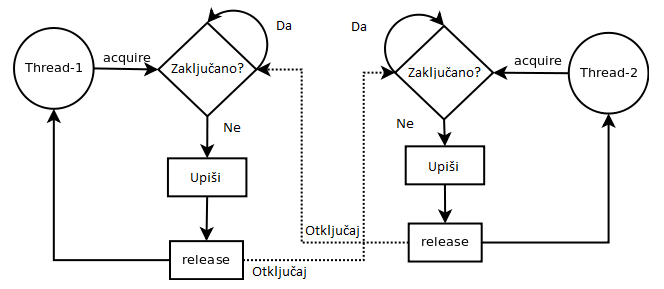
Ovim smo popravili kod i program i onemogućili da nam se javi greška ili da se dese ne očekivane aktivnosti u fajlu. Konačan izlaz konzole je prikazan na sledećoj slici.



Slika 5: output for lock

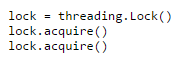
Na priloženoj slici možemo videti da se prvo poziva acquire metod Lock-a koji zaključava nit broj dva potom se obavlja upisivanje skinutog sadržaja sa priloženog linka, potom se poziva metod release() koji, kako je navedeno ranije, otključava ponovo bravu, te možemo aktivirati sledeću nit. Sličan tok akcija se u nastavku dešava do poslednje niti koja se izvrši. Napomena:

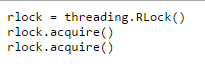
Svaki thread od dva skida sa dva različita sajta, odnosno dve niti ukupno skidaju sa četiri sajta.



Slika 6: Dijagram stanja

Logika iza Rlock-a (reentrant), brava koja se može više puta zaključati bez blokiranja, je da ta da nam omogući da više puta pozovemo metod aquire() od neke niti, bez blokiranja iste. Naravno, koliko puta pozovemo acquire() metodu i koliko puta zaključamo bravu toliko puta moramo pozvati i metod release() kako bi otključali bravu, i resurs. Ovo neće biti implementirano u projektu jer nema neke preke potrebe za tim te ću samo priložiti deo koga, kako bi izgledalo to u implementaciji.



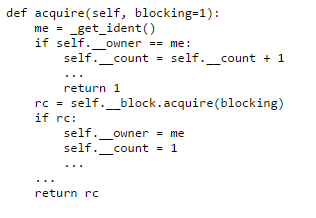


Slik 7: Rlock vs lock

Vidimo da je sintaksa identična kao i kod običnog zaključavanja.

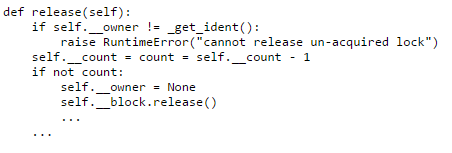
Ono što Rlock još koristi je thread.allocate\_lock(), međutim, on čuva stanje roditelja niti kako bi omogućio tu višeulaznost (reentrant).

Implementacija



Slika 8: Rlock implementacija – acquire()

Ukoliko je roditelj resurs, nit koja poziva metod acquire(), brojač se inkrementuje za jedan. Ukoliko to nije slučaj, pozvaće se acquire(). Prvi put kada se kada se zaključa brava, roditelj se čuva a brojač inicijalizuje na 1.



Slika 9: Rlock implementacija – release()

Rlock release() metod prvenstveno proverava da li je nit koja poziva metod roditelj brave. Ukoliko jeste, brojač se umanjuje, kada je brojač jednak 0, resurs je otključan te je dostupan ostalim nitima na korišćenje.

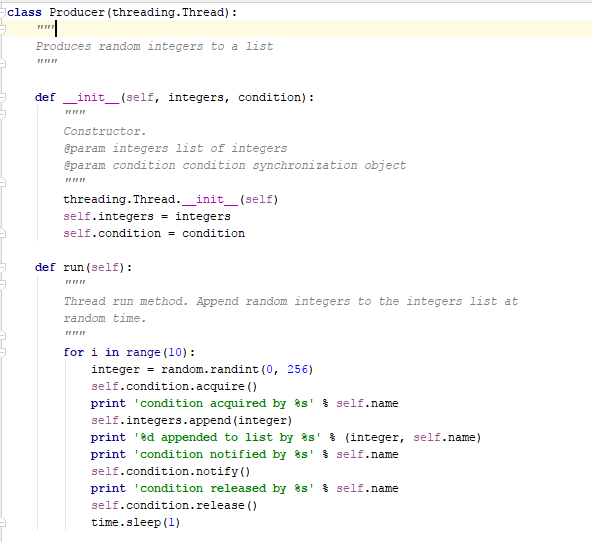
Uslovni mehanizam:

Uslovni mehanizam je sinhroni mehanizam gde nit čeka određeni uslov za koji druga nit signalizira da se desio. Kada se uslov desi, nit zaključava bravu, kako bi dobio ekskulizivn pristup deljenom resursu.

Kako bi se ovo što bolje objasnilo biće korišćen drugi primer koji je uzet sa interneta i dorađen radi anlize.

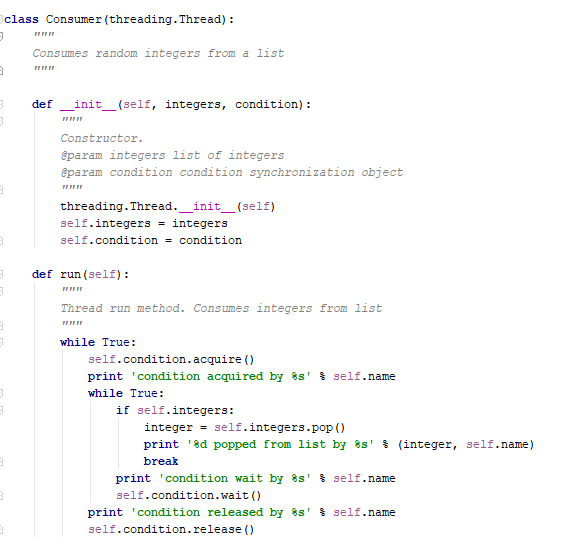
Problem koji će se implementirati je dobro poznati problem u računarskim naukama, konkretno u višenitnoj sinhronizaciji. Problem se sastoji u tome što imamo proizvođača i kupca, dok proizvođač proizvodi neki nasumičan (random) integer, kupac uzima taj integer. Ne sme se dozvoliti da proizvođač pravi nov proizvod dok se stari ne kupi, kao što i ne smemo dati kupcu da kupuje iz prazne korpe ukoliko proizvođač još nije proizveo naš proizvod. Rešenje za ovaj problem se ogleda u tome da možemo ili uspavati određenu nit, ili iz bafera izbaciti svaki dodatni resurs kada je on pun. Svaki put kada kupac uzme proizvod signaliziraće to proizvođaču, koji pristupa pravljenju novog proizvoda. Slično ovome ukoliko je naš bafer prazan možemo kupca uspavati dok se isti ne napuni, potom obaveštavamo kupca da je bafer put i da se može probuditi iz stanja spavanja i pokupiti taj resurs.

U konkretnoj implementaciji, dodajemo random integer u listu, u random vremenu, te kupac kupi te brojeve nakon što ubacimo u listu i obvestimo ga.



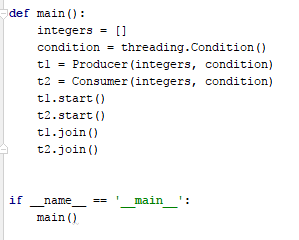
Slika 10: Producer/customer problem – Producer class

Klasaa Producer, zaključava bravu, dodaje integer, obaveštava nit kupca da postoji podatak koji treba da pokupi, i da otkljuca bravu. S obzirom da je beskonačna petlja ovo će se beskonačno i vrteti dok ne zustavimo program. Posle svake operacije implementirana je određena pauza pre nego što se nastavi dalje.



Slika 11: Producer/customer problem – Customer class

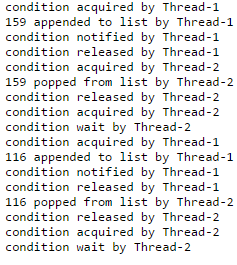
U samoj Customer klasi, slične se stvari desavaju. Zaključava se brava, proverava se da li postoji integer koji je potrebno pokupiti, ukoliko nema ništa, stavlja se na čekanje, dok kupac nit ne bude obveštena od strane prodavac niti. Kada se pojavi integer i obavesti se kupac, on kupi iz liste taj integer, i oslobađa bravu.



Slika 12: Producer/customer problem – Main class

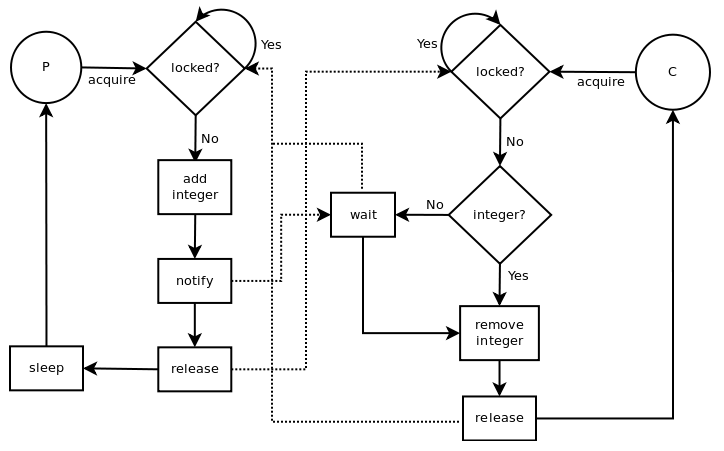
Main metoda se implementira tako što pravimo dve niti te ih nakon toga i pokrećemo.

Izlaz tj. Output programa je sledeći:



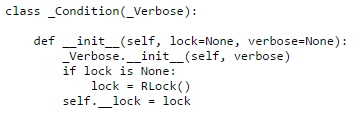
Slika 13: Output progrma Producer/Customer

Kao što je već objasnjeno nakon prethodnih slika, proizvođač upisuje neki nasumičan integer u listu jednom niti, a potom drugom niti, kupac, kupi taj integer i liste. Akcija se identično ponavlja beskonačno puta dok se ne prekine rad programa. Posle svakog ’’pravljena’’ integera i ’’kupljenja’’ istog, dešav se mala pauza, dakle, pauza između punog ciklusa. Ovim smo objasnili i implementirali rešenje poznatog problema, koji se koristi u operativnim sistemima i danas, kao i u raznim programima tj. Aplikacijama.



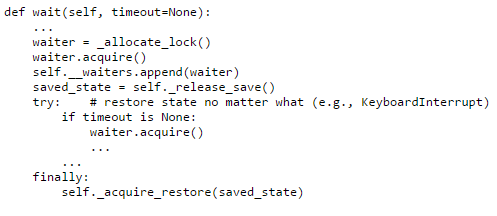
Slika 14: Use-case dijagram

Sada ćemo se osvrnuti dublje i malo detaljnije u sam mehanizam uslovne sinhronizacije.



Slika 15: Condition mechnism

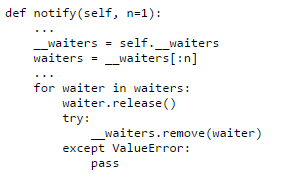
Konstruktor uslovne (Condition) sinhronizcije pravi rlock, ukoliko mu lock parametar nije prosleđen. Ovakav lock će kasnije biti korišćen za acquire i release metode.



Slika 16: Wait()

Sledeća je metoda wait. Recimo da je pozivamo sad bez neke vremenske vrednosti radi lakšeg objasnjenja. Pravimo lock pod nazivom waiter i njegovo stanje postavljamo na zaključano. Waiter bravu koristimo za komunikaciju između niti, tako da proizvođač može obavestiti kupca otključavajući (release) waiter-ovu bravu. Objekat brave se dodaje na listu waiter-a a metoda blokira u delu waiter.acquire(). Možemo videti i da se stanje brave čuva na početku a kasnije se vraća kada se wait() metoda izvrši i vrati povratnu (return) vrednost.

Konačno imamo notify metodu, koju se koristi kako bi otključali bravu waiter. Proizvođač takođe poziva notify() metod, da obavesti kupca da je blokiran na wait() metodi. Nadam se da sam uspeo koliko toliko da približim ovo jer mi je teško da ovako stručne izraze prenesem smisleno sa engleskog na srpski, jer u progrmiranju i životu koristim uglavnom strane termiine, i da bi se objsnili ovakvi koncepti na srpskom mora se odlično poznavati oblast, i biti ekspert u istoj, što ja svakako nisam.



Slika 17: Notify mehanizam.

Semafori:

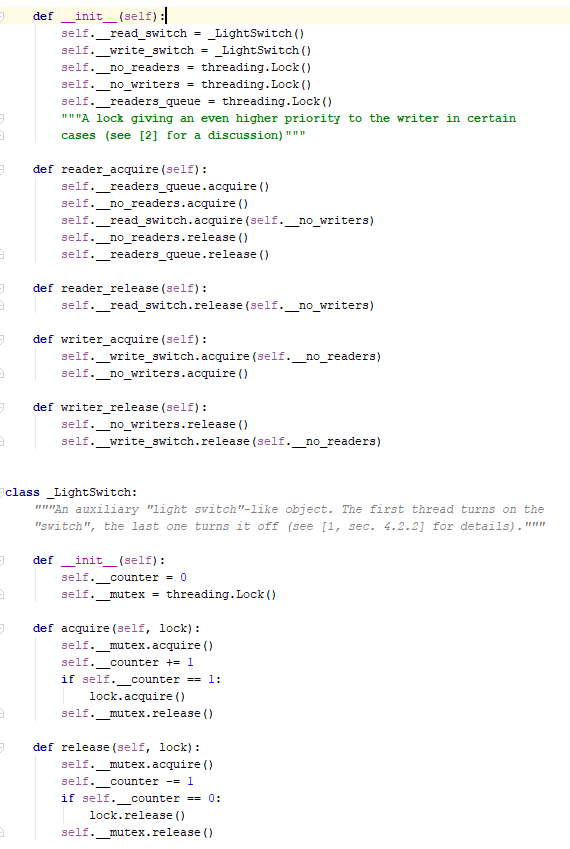
Koncept semafora je baziran na internom brojaču, koji se umanjuje svaki put kada se metoda acquire() pozove, a uvećava kada je release metoda pozvana(). Ukoliko se brojač nađe na 0, tada acquire metoda blokira nit.

Jednostavna ali efikasna implementacija koncepta semafor koju je čuveni Djikstra implementirao. Semafore uglavnom koristimo kada želimo da kontrolišemo pristup resursu zbog određenih limitacija, najjednostavniji primer bi bio server.

U nastavku će biti objasnjena i anlizirana implementacija autora Mteusz Kobos-a, koji je na vrlo efikasan način uspeo da reši naširoko poznat problem Reader-Writer koji je tako Djikstra postavio. Problem, tačnije rešenje problema je implementirano kroz python programski jezik. Dakle, problem koji će ovde biti analiziran je takozvani Drugi problem Pisca-čitača (The second reader writer problem). U ovom problemu, više čitalaca mogu istovremeno pristupiti deljenom resursu, dok pisac (writer) ima ekskluzivan (primaran) pristup tom deljenom resursu. Ovakvi problemi se najviše javljaju kod upisa velikog broja podataka u baze podataka, dok se u toku samog upisa mora dozvoliti ostalim korisnicima da ne smetano pristupaju i čitaju određeni resurs.

Možemo definisati i nekoliko uslova koji se moraju ispoštovati:

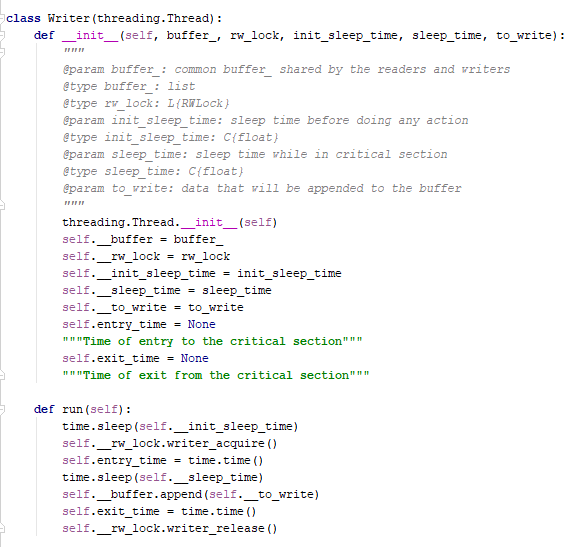
* Ni jedan čitač ne sme biti na čekanju ukoliko je resurs otvoren za čitanje, osim ukoliko pisac trenutno ne upisuje u resurs.
* Pisac se ne sme zadržati duže nego što je potrebno da izvrši svoju akciju.



Slika 18: Implementacija Reader/Writer problema

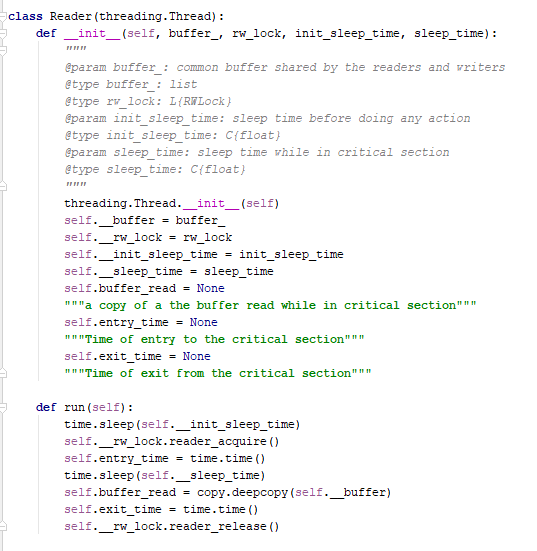
Autor prvenstveno implementira prekidač za oba, pisca i čitača. Nakon toga implementira brave (lock) za takođe pisca i čitača, tačnije za onda kada ih nema.

Sam prekidač (Lightswitch) radi na jednostavnom principu, gde prva nit uključuje prekidač a poslednja gasi svetlo za sobom. Ovde vidimo da se brojač uveća za 1 kada se pozove metoda acquire() a isto tako se smanji za jedan kada se pozove metoda release().



Slika 19: Pisac klasa

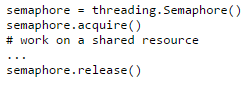
U konstruktoru klase pisca, prosleđujemo zajednički deljeni bafer između pisca i čitača, potom bravu (lock), inicijalno vreme kada treba proces da se pokrene, posle koliko vremena, vreme koje proces treba da provede u modu čekanja, i na kraju resurs koji će se upisati u bafer.



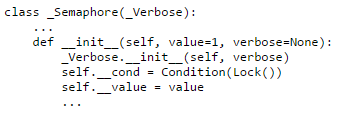
Slika 20: Čitač klasa

Identično kao i u klasi pisca autor koda implementira konstruktor sa deljenim baferom, bravom, vreme za koje treba da se odloži početak niti, i na kraju vreme koje nit treba da stoji u stanju čekanja. Primećujemo da nam nedostaje upisivanje u bafer, što je verovtno više nego očigledno, jer čitač ima samo mogućnost čitanja iz bafera, dok pisac može upisivati, jer mu je kako mu i sam naziv kaže to i posao.

Sami semfori u jeziku python ne vezano za postojeće probleme se pozivaju na sledeći veoma jednostavan način:

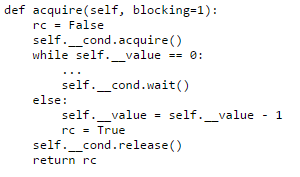


Slika 21: Semafori



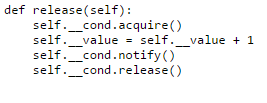
Slika 22: Klasa semafor

Sam mehanizam semafora radi tako što konstruktor prihvata vrednost koja se inicijalizuje a pored toga predstavljaa i brojač. Defultna, podešena, vrednost ovog brojača je 1. Kreira se uslovna instanca sa lockom kako bi se zaštitio brojač, i kako bi obavestili ostale niti kada je semafor kasnije otključan (released).



Slika 23: Semafor – acquire metoda

Sledeća je acquire() metoda. Kada je brojač smafora postavljen na 0, tj. Jednak 0, on se blokira metodom wait() dok ne bude obavešten od strane druge niti. Ukoliko je brojač semfora veći od 0, on umanjuje vrednost.

  
Slika 24: Semafor release metoda

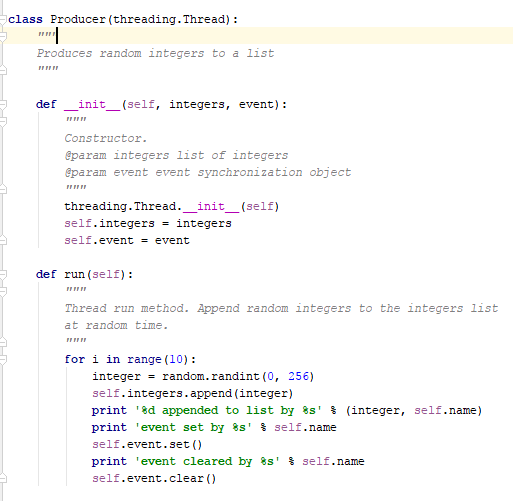
Jednostvn metoda koja povećava brojač za jedan, i obaveštava potom sve druge niti o tome.

Event (dogđaj) :

Veoma jednostavan mehanizam. Nit signalizira događaj a ostale niti potom čekaju na njega.

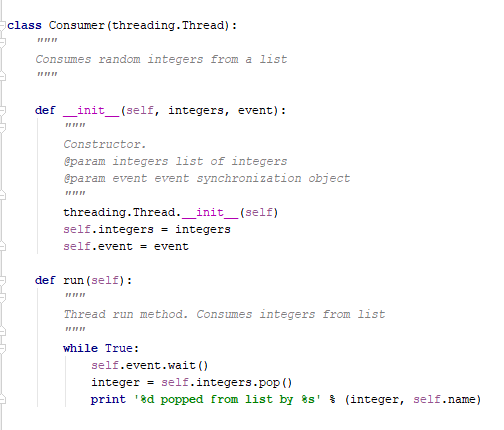
Kako bi ovo pokazali, vratićemo se na naš prethodno implementirn kod proizvođača i kupca, i umesto uslova, implementiraćemo događaj kako bi videli sta še i kako dešava.

Prvenstveno imamo klasu proizvođač:



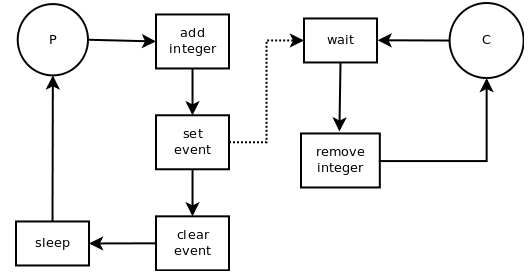
Slika 25: Proizvođač klasa – event

Prosleđujemo događaj (event) instancu konstruktoru umesto prethodno instanciranog stanja. Svaki put kada je integer dodat u listu, događaj je postavljen potom odmah i sklonjen kako bi obavestio kupca. Ovo sklanjanje tj. Čišćenje (clear) događaja se dešava defaultno.



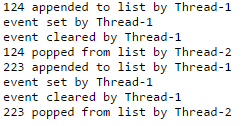
Slika 26: Consumer klasa

U ovog klasi mi takođe prosleđujemo Event (događaj) konstruktoru. Instanca kupca se blokira na wait() metodi, dok se događaj ne postavi (set), obaveštavajući ostale da postoji integer vrednost koja treba da se iskoristi.



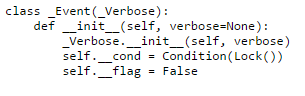
Slika 27: Dijagram stanja

Ovo se na priloženom dijagramu može odlično videti, i još bolje objasniti, te je isti i proložen zarad toga.



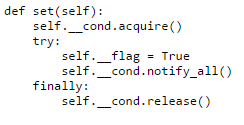
Slika 28: Izlaz u konzoli

Za sam kraj demonstracije događaja pogledaćemo detaljnije sam mehanizam događaja kako je implementiran u pythonu.



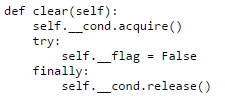
Slika 29: Event

Prvenstveno imamo konstruktor događaja. Kreira se instanca uslova sa bravom (lock) kako bi zaštitila vrednost posle ispaljivanja događaja, ali i da bi se obavestile ostale niti da je događaj postavljen.



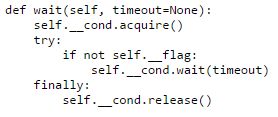
Slika 30: Set

Potom imamo set metodu, postavlja flag na true i obveštava ostale niti. Uslovni (condition) objekat se koristi kako bi se zaštitio kritični deo kada se vrednost flag-a promeni.



Slika 31: Clear

Clear metoda postavlja vrednost flag-a na false. Dakle, suporotna metoda od metode set().



Slika 32: Wait metod

Metoda koja služi za blokadu dok se set metoda ne pozove. With metoda ne radi ništa ukoliko je flag postavljen.

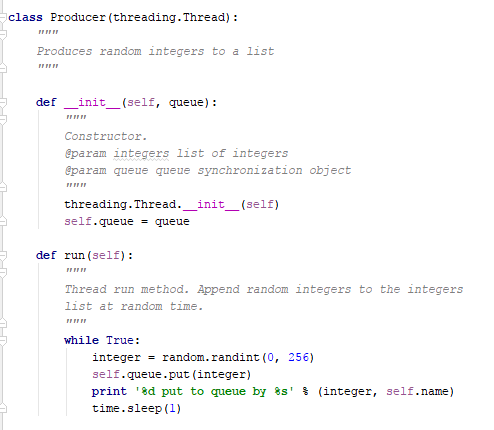
Queque (red):

Redovi su odličan mehanizam kada treba da razmenimo informacije između niti. U ovom slučaju redovi sami rešavaju problem zaključavanja, što nam dodatno olakšava posao.

Kod samih redova imamo četiri bitne metode:

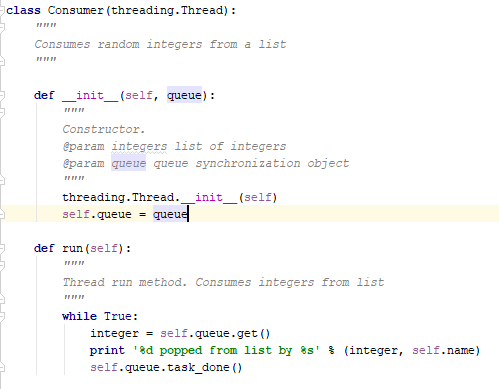
* Put: dodaje stavku u red
* Get: izbacuje i vraća stavku iz reda
* Task\_done: Poziva se svaki put kada je stavka procesuira
* Join: Blokira dok se sve stavke ne procesuiraju

Još jednom ćemo iskoristiti naš program Producer/customer, kako bi implementirali red u njega i objasnili detaljnije svrhu ovog mehanizma.



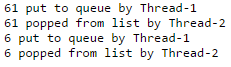
Slika 33: Producer queue

Prvo klasa proizvođač. Ovoga puta ne treba da prosledimo listu integer vrednosti, pošto koristimo red za čuvanje itegera koji se generišu. Nit generiše i ubacuje integere u red do beskonačnosti jer je implementiran beskonačan loop.



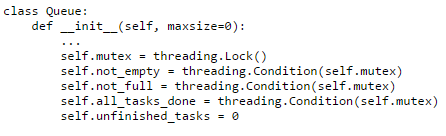
Slika 34: Customer queue klasa

U ovoj klasi, nit uzima integer vrednost iz reda, i nagoveštava da je ovo urađeno pomoću task\_done() metode.



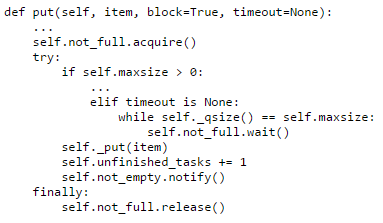
Slika 35: Izlaz progrma

Queue (red) module, sam brine u zaključavanju za nas što je svakako velika prednost. Da bi ovo videli još bolje osvrnućemo se detaljnije nas sam mehanizam redova koje implementira python. Dakle, kako ’’ispod’’ ovaj mehanizam radi.



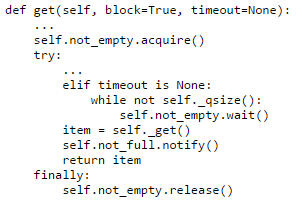
Slika 36: Red (Queue)

Queue konstruktor kreira lock da bi zaštitio red kada se elementi dodaju ili uklanjaju. Neki uslovni objekti se takođe kreiraju kako bi nas obavestili o događajima tipa red nije prazan (queue is not empty), red nije put, i slično.



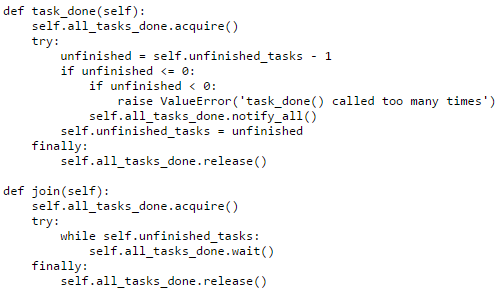
Slika 37: Put

Put metoda dodaje stavku ili čeka ukoliko je red pun. Ona obaveštava niti blokirane na get() metodi, da red nije prazan.



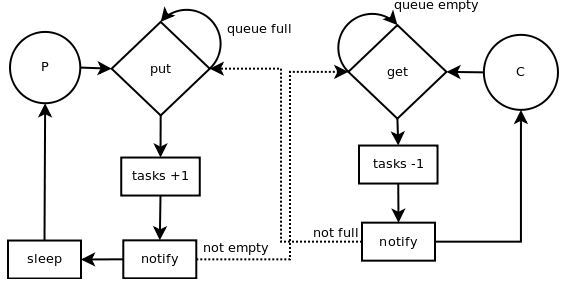
Slika 38: Get

Get metoda izbacuje stavku iz reda ili čeka ukoliko je red prazan. Takođe obaveštava niti blokirane na put() metodi, da red nije pun.



Slika 39: Task done

Kada je metoda task done pozvana, broj nezavršenih taskova se smanjuje. Ukoliko je brojač na 0, onda niti čekaju na metodi join kako bi nastavili njihovo izvršavanje.



Slika 40: Dijagram stanja

# Zaključak:

Pre samog početka izrade projekta nisam imao velika očekivanja niti od mene niti od projekta, jednostavno nije postojala konkretna ideja. Posle razgovora i sugestija asistenta javil se želja i motivacija da se obradi gore obrađena tema. Kroz ovaj projekat sam prvenstveno popravio greške iz nekih drugih projekata koje su bile vezane za niti i konkuretne procese, što mi je izuzetno drago, jer programi više ne pucaju nasumično, što dalje dovodi do zaključka da su teme obrađene u ovom projektu obrađene uspesno, i da je student naučio i stekao nova znanja što je bio i cilj projekta. U budućnosti, ukoliko budem radio sa nitima neku aplikaciju ili sistem što definitivno i svakako hoću, uvek ću se osvrnuti na to koliko sam bio ponosan kada sam u toku izrade projekta i istraživanja same teme pronašao odgovore na druge probleme iz drugih programa.

# Reference:

[1] A.B. Downey: "The little book of semaphores", Version 2.1.5, 2008

[2] P.J. Courtois, F. Heymans, D.L. Parnas:

"Concurrent Control with 'Readers' and 'Writers'",

Communications of the ACM, 1971 (via [3])

[3] <http://en.wikipedia.org/wiki/Readers-writers_problem>

[4] <http://code.activestate.com/recipes/577803-reader-writer-lock-with-priority-for-writers/>

[5] <https://stackoverflow.com/>

[6] <https://docs.python.org/2/>

[7] lams.metropolitan.ac.rs Predavanja i vežbe