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.

FetchUrls

```
1  import threading
2  import urllib2
3
4  class FetchUrls(threading.Thread):
5      """
6      Thread checking URLs.
7      """
8      def __init__(self, urls, output):
9          """
10         Constructor.
11         @param urls list of urls to check
12         @param output file to write urls output
13         """
14         threading.Thread.__init__(self)
15         self.urls = urls
16         self.output = output
17
18     def run(self):
19         """
20         Thread run method. Check URLs one by one.
21         """
22         while self.urls:
23             url = self.urls.pop()
24             req = urllib2.Request(url)
25             try:
26                 d = urllib2.urlopen(req)
27             except urllib2.URLError, e:
28                 print 'URL %s failed: %s' % (url, e.reason)
29             self.output.write(d.read())
30             print 'write done by %s' % self.name
31             print 'URL %s fetched by %s' % (url, self.name)
32
33 def main():
34     # list 1 of urls to fetch
35     urls1 = ['http://www.google.com', 'http://www.facebook.com']
36     # list 2 of urls to fetch
37     urls2 = ['http://www.yahoo.com', 'http://www.youtube.com']
38     f = open('output.txt', 'w+')
39     t1 = FetchUrls(urls1, f)
40     t2 = FetchUrls(urls2, f)
41     t1.start()
42     t2.start()
43     t1.join()
44     t2.join()
45     f.close()
46
47 if __name__ == '__main__':
48     main()
```

Slika 1: Niti

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

```
C:\Python27\python.exe C:/Users/ryz/Desktop/CS225-Projekat/lock.py
write done by Thread-1
URL http://www.facebook.com fetched by Thread-1
write done by Thread-1
URL http://www.google.com fetched by Thread-1
write done by Thread-2
URL http://www.youtube.com fetched by Thread-2
write done by Thread-2
URL http://www.yahoo.com fetched by Thread-2

Process finished with exit code 0
```

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:

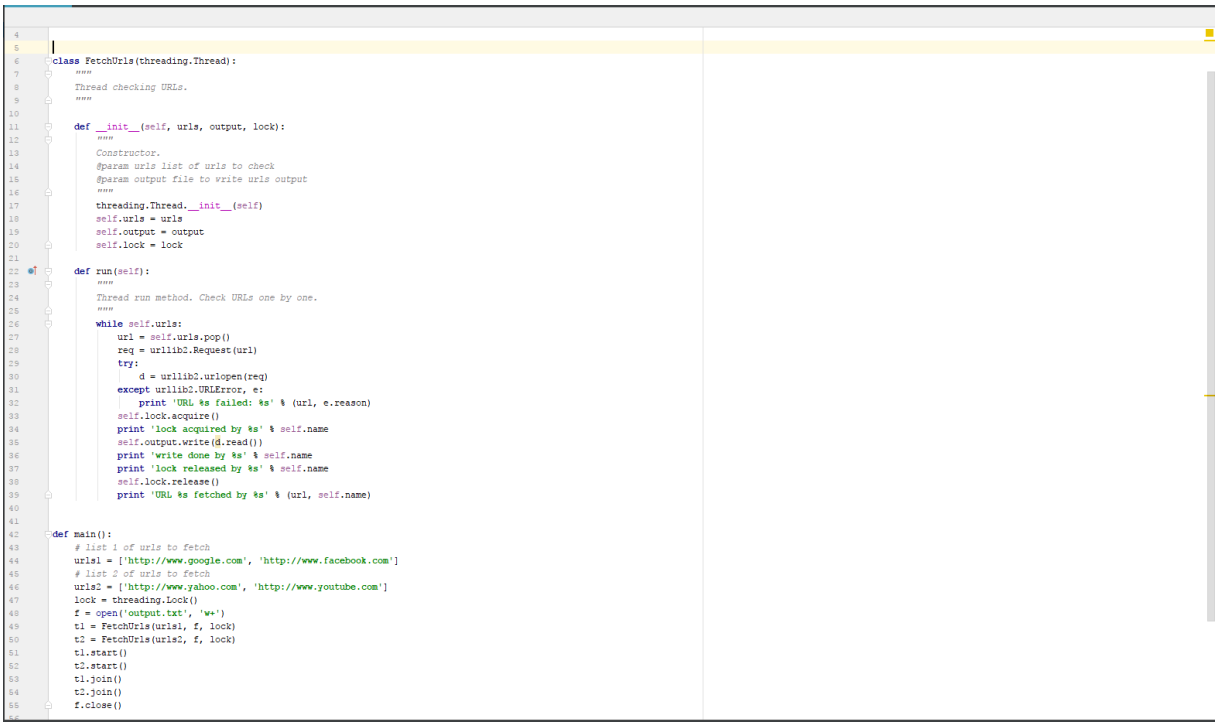
*Tačkice predstavljaju prethodno implementiran kod koji nećemo u ovom delu ponoviti, te samo prikazujemo novo imlementirani.

```
class FetchUrls(threading.Thread):
    ...

    def __init__(self, urls, output, lock):
        ...
        self.lock = lock

    def run(self):
        ...
        while self.urls:
            ...
            self.lock.acquire()
            print 'lock acquired by %s' % self.name
            self.output.write(d.read())
            print 'write done by %s' % self.name
            print 'lock released by %s' % self.name
            self.lock.release()
            ...


def main():
    ...
    lock = threading.Lock()
    ...
    t1 = FetchUrls(urls1, f, lock)
    t2 = FetchUrls(urls2, f, lock)
    ...
```

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.



```
class FetchUrls(threading.Thread):
    """
    Thread checking URLs.
    """

    def __init__(self, urls, output, lock):
        """
        Constructor.
        @param urls list of urls to check
        @param output file to write urls output
        """
        threading.Thread.__init__(self)
        self.urls = urls
        self.output = output
        self.lock = lock

    def run(self):
        """
        Thread run method. Check URLs one by one.
        """
        while self.urls:
            url = self.urls.pop()
            req = urllib2.Request(url)
            try:
                d = urllib2.urlopen(req)
            except urllib2.URLError, e:
                print 'URL %s failed: %s' % (url, e.reason)
            self.lock.acquire()
            print 'lock acquired by %s' % self.name
            self.output.write(d.read())
            print 'write done by %s' % self.name
            print 'lock released by %s' % self.name
            self.lock.release()
            print 'URL %s fetched by %s' % (url, self.name)


def main():
    # list 1 of urls to fetch
    urls1 = ['http://www.google.com', 'http://www.facebook.com']
    # list 2 of urls to fetch
    urls2 = ['http://www.yahoo.com', 'http://www.youtube.com']
    lock = threading.Lock()
    f = open('output.txt', 'w+')
    t1 = FetchUrls(urls1, f, lock)
    t2 = FetchUrls(urls2, f, lock)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    f.close()
```

Slika 4: Threads and Lock

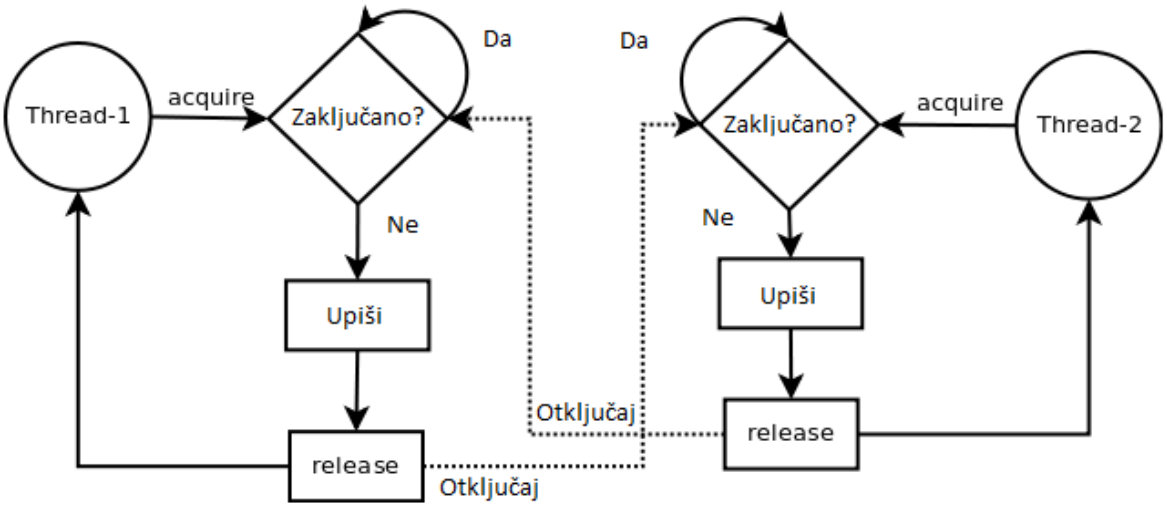
Ovim smo popravili kod i program i onemogućili da nam se javi greška ili da se dese ne očekivane aktivnosti u fajlu. Konačan izlaz konzole je prikazan na sledećoj slici.

```
C:\Python27\python.exe C:/Users/ryz/Desktop/CS225-Projekat/lock.py
lock acquired by Thread-2
write done by Thread-2
lock released by Thread-2
URL http://www.youtube.com fetched by Thread-2
 lock acquired by Thread-1
write done by Thread-1
lock released by Thread-1
URL http://www.facebook.com fetched by Thread-1
lock acquired by Thread-1
write done by Thread-1
lock released by Thread-1
URL http://www.google.com fetched by Thread-1
lock acquired by Thread-2
write done by Thread-2
lock released by Thread-2
URL http://www.yahoo.com fetched by Thread-2

Process finished with exit code 0
```

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.

```
lock = threading.Lock()
lock.acquire()
lock.acquire()
```

```
rlock = threading.RLock()
rlock.acquire()
rlock.acquire()
```

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

```
def acquire(self, blocking=1):
    me = _get_ident()
    if self.__owner == me:
        self.__count = self.__count + 1
        ...
        return 1
    rc = self.__block.acquire(blocking)
    if rc:
        self.__owner = me
        self.__count = 1
        ...
    ...
    return rc
```

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.

```
def release(self):
    if self.__owner != _get_ident():
        raise RuntimeError("cannot release un-acquired lock")
    self.__count = count = self.__count - 1
    if not count:
        self.__owner = None
        self.__block.release()
        ...
    ...
```

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.

```
class Producer(threading.Thread):
    """
    Produces random integers to a list
    """

    def __init__(self, integers, condition):
        """
        Constructor.
        @param integers list of integers
        @param condition condition synchronization object
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.condition = condition

    def run(self):
        """
        Thread run method. Append random integers to the integers list at
        random time.
        """
        for i in range(10):
            integer = random.randint(0, 256)
            self.condition.acquire()
            print 'condition acquired by %s' % self.name
            self.integers.append(integer)
            print '%d appended to list by %s' % (integer, self.name)
            print 'condition notified by %s' % self.name
            self.condition.notify()
            print 'condition released by %s' % self.name
            self.condition.release()
            time.sleep(1)
```

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.

```
class Consumer(threading.Thread):
    """
    Consumes random integers from a list
    """

    def __init__(self, integers, condition):
        """
        Constructor.
        @param integers list of integers
        @param condition condition synchronization object
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.condition = condition

    def run(self):
        """
        Thread run method. Consumes integers from list
        """
        while True:
            self.condition.acquire()
            print 'condition acquired by %s' % self.name
            while True:
                if self.integers:
                    integer = self.integers.pop()
                    print '%d popped from list by %s' % (integer, self.name)
                    break
                print 'condition wait by %s' % self.name
                self.condition.wait()
            print 'condition released by %s' % self.name
            self.condition.release()
```

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.

```
def main():
    integers = []
    condition = threading.Condition()
    t1 = Producer(integers, condition)
    t2 = Consumer(integers, condition)
    t1.start()
    t2.start()
    t1.join()
    t2.join()


if __name__ == '__main__':
    main()
```

Slika 12: Producer/customer problem – Main class

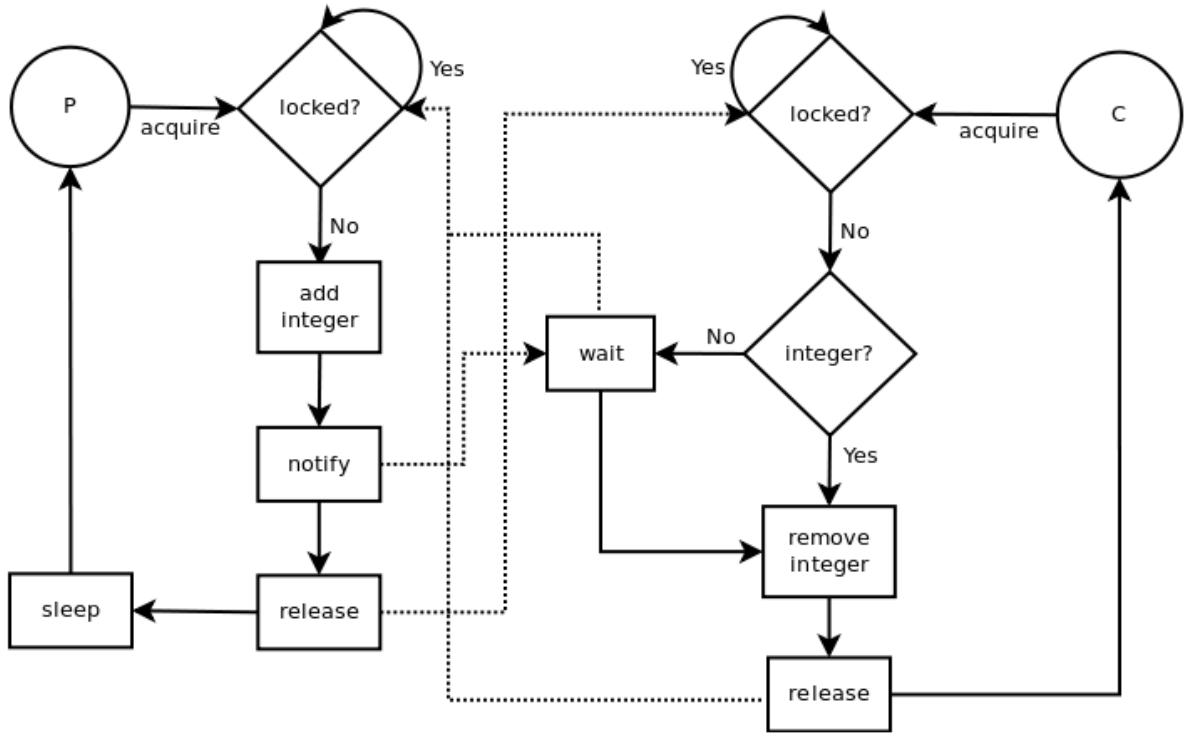
Main metoda se implementira tako što pravimo dve niti te ih nakon toga i pokrećemo.

Izlaz tj. Output programa je sledeći:

```
condition acquired by Thread-1
159 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
condition acquired by Thread-2
159 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
condition acquired by Thread-1
116 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
116 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
```

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.

```
class _Condition(_Verbose):

    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

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.

```
def wait(self, timeout=None):
    ...
    waiter = _allocate_lock()
    waiter.acquire()
    self.__waiters.append(waiter)
    saved_state = self._release_save()
    try:    # restore state no matter what (e.g., KeyboardInterrupt)
        if timeout is None:
            waiter.acquire()
            ...
        ...
    finally:
        self._acquire_restore(saved_state)
```

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.

```
def notify(self, n=1):
    ...
    __waiters = self.__waiters
    waiters = __waiters[:n]
    ...
    for waiter in waiters:
        waiter.release()
        try:
            __waiters.remove(waiter)
        except ValueError:
            pass
```

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.

```
    def __init__(self):
        self.__read_switch = _LightSwitch()
        self.__write_switch = _LightSwitch()
        self.__no_readers = threading.Lock()
        self.__no_writers = threading.Lock()
        self.__readers_queue = threading.Lock()
        """A lock giving an even higher priority to the writer in certain
        cases (see [2] for a discussion)"""

    def reader_acquire(self):
        self.__readers_queue.acquire()
        self.__no_readers.acquire()
        self.__read_switch.acquire(self.__no_writers)
        self.__no_readers.release()
        self.__readers_queue.release()

    def reader_release(self):
        self.__read_switch.release(self.__no_writers)

    def writer_acquire(self):
        self.__write_switch.acquire(self.__no_readers)
        self.__no_writers.acquire()

    def writer_release(self):
        self.__no_writers.release()
        self.__write_switch.release(self.__no_readers)


class _LightSwitch:
    """An auxiliary "light switch"-like object. The first thread turns on the
    "switch", the last one turns it off (see [1, sec. 4.2.2] for details)."""

    def __init__(self):
        self.__counter = 0
        self.__mutex = threading.Lock()

    def acquire(self, lock):
        self.__mutex.acquire()
        self.__counter += 1
        if self.__counter == 1:
            lock.acquire()
        self.__mutex.release()

    def release(self, lock):
        self.__mutex.acquire()
        self.__counter -= 1
        if self.__counter == 0:
            lock.release()
        self.__mutex.release()
```

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().

```
class Writer(threading.Thread):
    def __init__(self, buffer_, rw_lock, init_sleep_time, sleep_time, to_write):
        """
        @param buffer_: common buffer_ shared by the readers and writers
        @type buffer_: list
        @type rw_lock: L{RWLock}
        @param init_sleep_time: sleep time before doing any action
        @type init_sleep_time: C{float}
        @param sleep_time: sleep time while in critical section
        @type sleep_time: C{float}
        @param to_write: data that will be appended to the buffer
        """
        threading.Thread.__init__(self)
        self.__buffer = buffer_
        self.__rw_lock = rw_lock
        self.__init_sleep_time = init_sleep_time
        self.__sleep_time = sleep_time
        self.__to_write = to_write
        self.entry_time = None
        """Time of entry to the critical section"""
        self.exit_time = None
        """Time of exit from the critical section"""

    def run(self):
        time.sleep(self.__init_sleep_time)
        self.__rw_lock.writer_acquire()
        self.entry_time = time.time()
        time.sleep(self.__sleep_time)
        self.__buffer.append(self.__to_write)
        self.exit_time = time.time()
        self.__rw_lock.writer_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.

```
class Reader(threading.Thread):
    def __init__(self, buffer_, rw_lock, init_sleep_time, sleep_time):
        """
        @param buffer_: common buffer shared by the readers and writers
        @type buffer_: list
        @type rw_lock: L{RWLock}
        @param init_sleep_time: sleep time before doing any action
        @type init_sleep_time: C{float}
        @param sleep_time: sleep time while in critical section
        @type sleep_time: C{float}
        """
        threading.Thread.__init__(self)
        self.__buffer = buffer_
        self.__rw_lock = rw_lock
        self.__init_sleep_time = init_sleep_time
        self.__sleep_time = sleep_time
        self.buffer_read = None
        """a copy of a the buffer read while in critical section"""
        self.entry_time = None
        """Time of entry to the critical section"""
        self.exit_time = None
        """Time of exit from the critical section"""

    def run(self):
        time.sleep(self.__init_sleep_time)
        self.__rw_lock.reader_acquire()
        self.entry_time = time.time()
        time.sleep(self.__sleep_time)
        self.buffer_read = copy.deepcopy(self.__buffer)
        self.exit_time = time.time()
        self.__rw_lock.reader_release()
```

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:

```
semaphore = threading.Semaphore()
semaphore.acquire()
# work on a shared resource
...
semaphore.release()
```

Slika 21: Semafori

```
class _Semaphore(_Verbose):
    ...
    def __init__(self, value=1, verbose=None):
        _Verbose.__init__(self, verbose)
        self.__cond = Condition(Lock())
        self.__value = value
        ...
```

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).

```
def acquire(self, blocking=1):
    rc = False
    self.__cond.acquire()
    while self.__value == 0:
        ...
        self.__cond.wait()
    else:
        self.__value = self.__value - 1
        rc = True
    self.__cond.release()
    return rc
```

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.

```
def release(self):
    self.__cond.acquire()
    self.__value = self.__value + 1
    self.__cond.notify()
    self.__cond.release()
```

Slika 24: Semafor release metoda

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

Event (događaj) :

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:

```
class Producer(threading.Thread):
    """
    Produces random integers to a list
    """

    def __init__(self, integers, event):
        """
        Constructor.
        @param integers list of integers
        @param event event synchronization object
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.event = event

    def run(self):
        """
        Thread run method. Append random integers to the integers list
        at random time.
        """
        for i in range(10):
            integer = random.randint(0, 256)
            self.integers.append(integer)
            print '%d appended to list by %s' % (integer, self.name)
            print 'event set by %s' % self.name
            self.event.set()
            print 'event cleared by %s' % self.name
            self.event.clear()
```

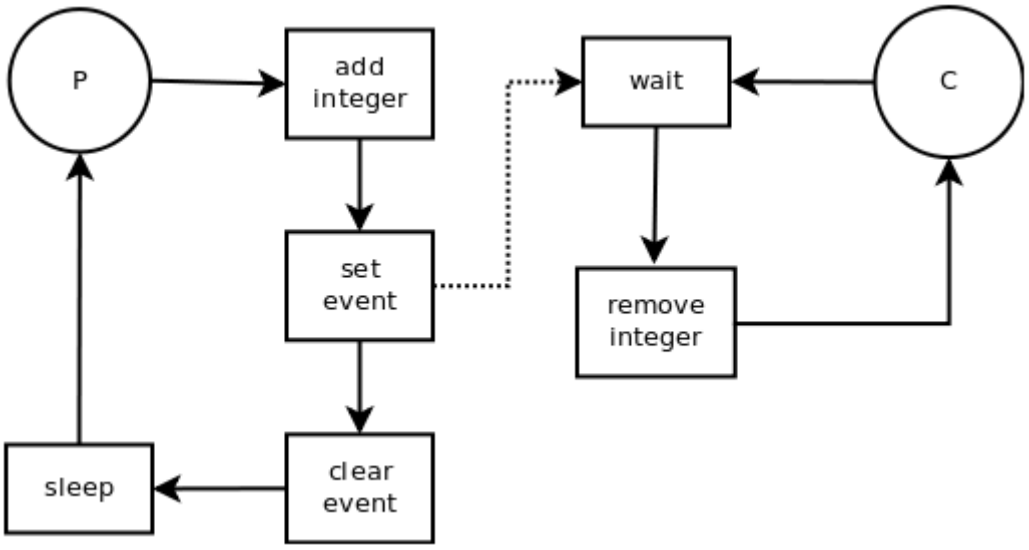
Slika 25: Proizvođač klasa – event

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.

```
class Consumer(threading.Thread):
    """
    Consumes random integers from a list
    """

    def __init__(self, integers, event):
        """
        Constructor.
        @param integers list of integers
        @param event event synchronization object
        """
        threading.Thread.__init__(self)
        self.integers = integers
        self.event = event

    def run(self):
        """
        Thread run method. Consumes integers from list
        """
        while True:
            self.event.wait()
            integer = self.integers.pop()
            print '%d popped from list by %s' % (integer, self.name)
```

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.

```
124 appended to list by Thread-1
event set by Thread-1
event cleared by Thread-1
124 popped from list by Thread-2
223 appended to list by Thread-1
event set by Thread-1
event cleared by Thread-1
223 popped from list by Thread-2
```

Slika 28: Izlaz u konzoli

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

```
class _Event(_Verbose):
    def __init__(self, verbose=None):
        _Verbose.__init__(self, verbose)
        self.__cond = Condition(Lock())
        self.__flag = False
```

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.

```
def set(self):
    self.__cond.acquire()
    try:
        self.__flag = True
        self.__cond.notify_all()
    finally:
        self.__cond.release()
```

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.

```
def clear(self):
    self.__cond.acquire()
    try:
        self.__flag = False
    finally:
        self.__cond.release()
```

Slika 31: Clear

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

```
def wait(self, timeout=None):
    self.__cond.acquire()
    try:
        if not self.__flag:
            self.__cond.wait(timeout)
    finally:
        self.__cond.release()
```

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.

```
class Producer(threading.Thread):
    """
    Produces random integers to a list
    """

    def __init__(self, queue):
        """
        Constructor.
        @param integers list of integers
        @param queue queue synchronization object
        """
        threading.Thread.__init__(self)
        self.queue = queue

    def run(self):
        """
        Thread run method. Append random integers to the integers
        list at random time.
        """
        while True:
            integer = random.randint(0, 256)
            self.queue.put(integer)
            print '%d put to queue by %s' % (integer, self.name)
            time.sleep(1)
```

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.

```
class Consumer(threading.Thread):
    """
    Consumes random integers from a list
    """

    def __init__(self, queue):
        """
        Constructor.
        @param integers list of integers
        @param queue queue synchronization object
        """
        threading.Thread.__init__(self)
        self.queue = queue

    def run(self):
        """
        Thread run method. Consumes integers from list
        """
        while True:
            integer = self.queue.get()
            print '%d popped from list by %s' % (integer, self.name)
            self.queue.task_done()
```

Slika 34: Customer queue klasa

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

```
61 put to queue by Thread-1
61 popped from list by Thread-2
6 put to queue by Thread-1
6 popped from list by Thread-2
```

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.

```
class Queue:
    def __init__(self, maxsize=0):
        ...
        self.mutex = threading.Lock()
        self.not_empty = threading.Condition(self.mutex)
        self.not_full = threading.Condition(self.mutex)
        self.all_tasks_done = threading.Condition(self.mutex)
        self.unfinished_tasks = 0
```

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.

```
def put(self, item, block=True, timeout=None):
    ...
    self.not_full.acquire()
    try:
        if self.maxsize > 0:
            ...
            elif timeout is None:
                while self._qsize() == self.maxsize:
                    self.not_full.wait()
        self._put(item)
        self.unfinished_tasks += 1
        self.not_empty.notify()
    finally:
        self.not_full.release()
```

Slika 37: Put

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

```
def get(self, block=True, timeout=None):
    ...
    self.not_empty.acquire()
    try:
        ...
        elif timeout is None:
            while not self._qsize():
                self.not_empty.wait()
        item = self._get()
        self.not_full.notify()
        return item
    finally:
        self.not_empty.release()
```

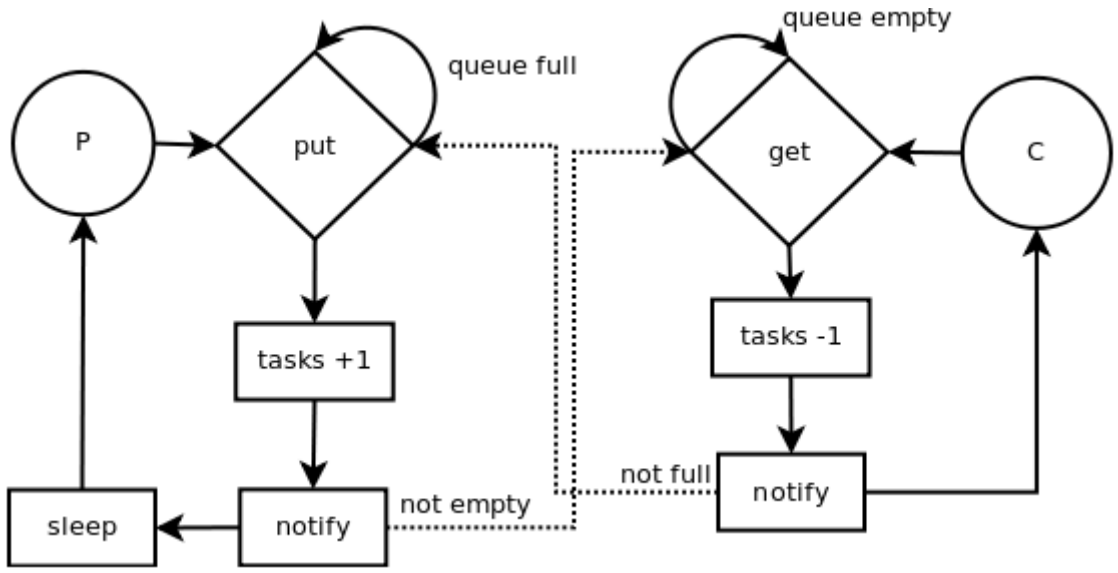
Slika 38: Get

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

```
def task_done(self):
    self.all_tasks_done.acquire()
    try:
        unfinished = self.unfinished_tasks - 1
        if unfinished <= 0:
            if unfinished < 0:
                raise ValueError('task_done() called too many times')
            self.all_tasks_done.notify_all()
        self.unfinished_tasks = unfinished
    finally:
        self.all_tasks_done.release()

def join(self):
    self.all_tasks_done.acquire()
    try:
        while self.unfinished_tasks:
            self.all_tasks_done.wait()
    finally:
        self.all_tasks_done.release()
```

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