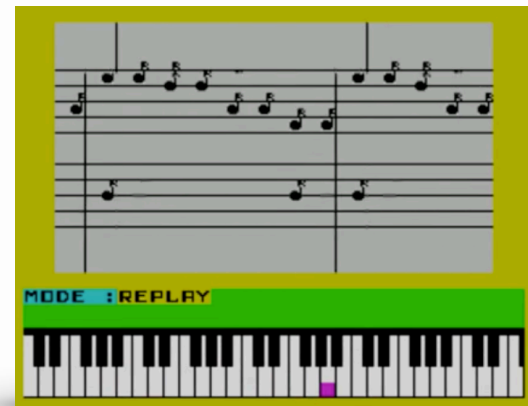


LINK*group*

Distance Learning System



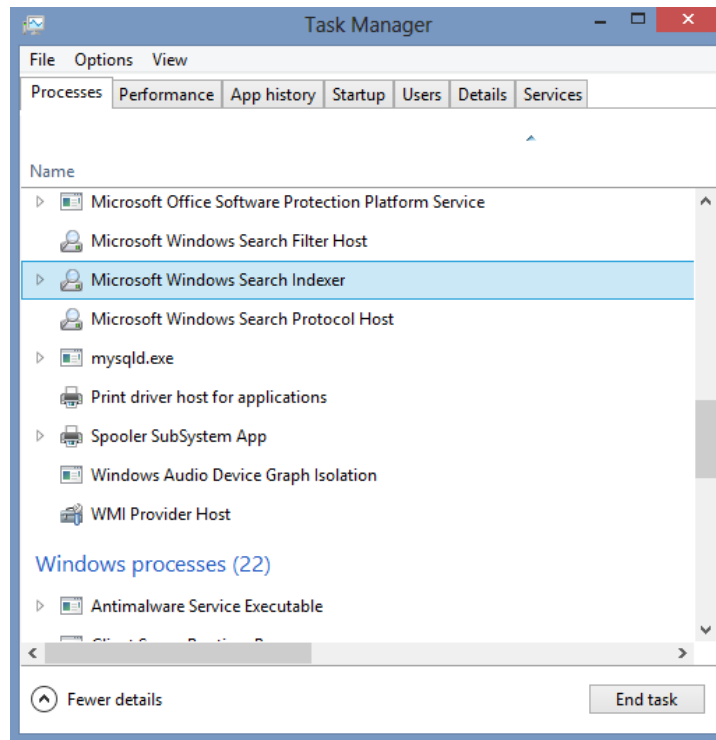
<https://www.youtube.com/watch?v=oNDVJLma-W4>

Niti

Python Network Programming

Procesi

- Iako grafički operativni sistem strpljivo čeka našu interakciju nakon startovanja, to ne znači da za to vreme ne radi ništa
- U stanju mirovanja, operativni sistem pokreće i kontroliše mnoštvo pozadinskih **procesa**
- Ovi procesi se izvršavaju paralelno, pa je rad korisnika neometan zato što svaki od njih ima zaseban adresni prostor i uzajamno nedostupne setove podataka



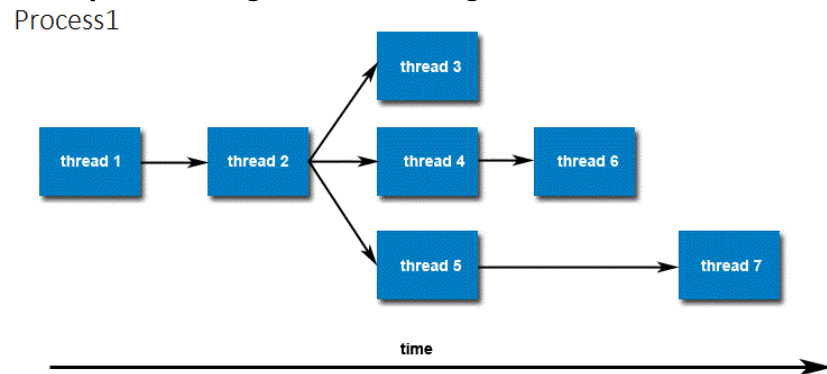
Procesi vs niti

- Postoje dva različita tipa **multitaskinga**
 - Multitasking zasnovan na procesima
 - Multitasking zasnovan na nitima (threads)
- Ukoliko sistem poseduje samo jedan procesor (odnosno procesor sa jednim jezgrom – single core), u jednom trenutku procesor može izvršavati samo jednu radnju, tako da se konkurencija postiže podelom procesorskog vremena među procesima (interapt).
- Danas su ipak u većini slučajeva zastupljeni procesori sa više jezgara, odnosno više procesora, tako da je moguće postići i realnu konkurentnost, i na taj način značajno ubrzati izvršavanje programa.



Procesi vs niti

- Procesi se najčešće smatraju sinonimom za aplikaciju ili program
- Ipak, ono što korisnik vidi kao jednu aplikaciju može u stvarnosti biti sastavljeno iz nekoliko procesa
- Svaki proces poseduje sopstveni memorijski prostor
- Nit predstavlja najmanju sekvencu programskih instrukcija kojom se može rukovati nezavisno.
- Nit(i) postoje u okviru procesa i svaki proces poseduje makar jednu nit. Glavna nit u programu se naziva **main thread**.
- Niti dele iste resurse, odnosno dele resurse procesa.
- Svaka nit poseduje mogućnost da kreira jednu ili više drugih niti.



Konkurencija i resursi

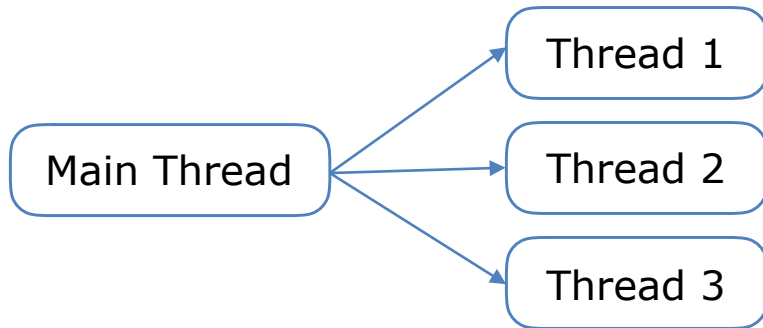
- Činjenica da niti dele isti memorijski prostor, pored pogodnosti, donosi i određene probleme. A to je konkurencija, odnosno pravo na određene resurse.
- Kada se jedna nit startuje, ona izvršava neki proces i najčešće manipuliše nekim resursom. Problem nastaje kada se startuje neka druga nit koja pokušava da pristupi istom tom resursu. U tom slučaju, i pored toga što se procesi izvršavaju paralelno, s obzirom na to da je resurs samo jedan, dolazi do problema u konkurenciji. Odnosno, jedna nit može blokirati resurs tako da druge niti ne mogu da ga koriste.
- Ovaj slučaj naziva se **Deadlock**.
- Deadlock se obično pojavljuje prilikom blokiranja resursa od strane više konkurentnih niti i često rezultira blokiranjem aplikacije.

Konkurencija i resursi

- Takođe, nit **ne mora** blokirati resurs.
- Tada, obe niti modifikuju resurs koji dobija vrednost koju mu je dodelila nit koja je poslednja na listi izvršavanja.
- Ovakav slučaj, u kome se niti *takmiče* za pristup resursu naziva se **Race Condition**.
- Kada startujemo nekoliko niti, nije sigurno da će one biti kompletno izvršavane prema redosledu startovanja. Svaka od njih dobiće određeni trenutak procesorskog vremena i taj trenutak iskoristiti za određenu radnju. U slučaju da iz nekog razloga ne uspe, procesorsko vreme će biti dodeljeno drugoj niti, koja će možda uspeti da izvrši radnju u zadatom roku. Na taj način, dobija se neočekivan rezultat. Na primer, ako u programu postoji promenljiva *x* i program se sastoji od sledećih niti:
 - Nit 1: dodeliti promenljivoj *x* vrednost 1
 - Nit 2: dodeliti promenljivoj *x* vrednost 2
 - Nit 3: dodeliti promenljivoj *x* vrednost 3
- nije garantovano da će na kraju programa *x* imati vrednost 3.
- Da bismo zagarantovali pravilan redosled izvršavanja niti, potrebno je da koristimo tehnike uzajamne komunikacije između niti, odnosno **signalizacije**. Na primer, možemo reći svim nitima da sačekaju izvršavanje jedne niti i slično.

Niti u Python-u

- Svaki program sastoji se bar od jedne niti. To je glavna nit, odnosno **main thread**. Ovu nit startuje Python virtualna mašina. Ona otvara nit za određeni program i izvršava njegov kod.
- Kada se glavna nit aktivira, ona može aktivirati neograničen broj niti unutar sopstvenog procesa.
Kada kažemo da može aktivirati neograničen broj niti, ne mislimo bukvalno, jer, iako ne postoji ograničenje u broju aktiviranih niti, ono se nameće ograničenjima samog okruženja (operativni sistem, memorija, hardver).



Niti u Pythonu

- Niti su nezaobilazna komponenta u Java-i u skoro svim oblastima: rukovanju grafičkim okruženjem, rukovanju ulazom i izlazom, mrežom...
- Teško je čak i zamisliti grafičku aplikaciju sa nekoliko tekst boksova koja će biti blokirana sve dok ne unesemo vrednosti u sve boksove (kao što bi bio slučaj u jednonitnoj konzolnoj aplikaciji).
- Rukovanje Socket-ima takođe je nezamislivo bez niti kako bi se ostvario neblokirani soket

Višejezgreni procesori i Python niti

- Više godina unazad, standard su multiprocesorske mašine, odnosno višejezgreni procesori.
- Ovakva arhitektura omogućava i paralelno izvršavanje procesa i Python virtualna mašina ima mogućnost iskorišćavanja ove pogodnosti prilikom manipulacije nitima.
- Paralelno izvršavanje u multiprocesorskom sistemu ne treba mešati sa paralelnim izvršavanjem niti.
- Niti čine kompaktan i nezavistan sistem paralelnog izvršavanja koji se može izvršavati na jednom ili više procesora. Dakle, nezavisne su od broja procesora i procesorske arhitekture.

Rukovanje nitima - Kreiranje niti

- Postoje dva načina za rad sa nitima u Pythonu. To su:
 - Korišćenje modula **_thread**
 - Korišćenje modula **threading**
 - Korišćenje modula **concurrent**
- Modul **_thread** omogućava operacije niskog nivoa, i bazira se na proceduralnom konceptu
- Modul **threading** je višeg nivoa, i omogućava klasni pristup nitima
- Modul **concurrent.futures** omogućava upravljano višenitno izvršavanje kao i čuvanje niti

Kreiranje niti pomoću modula `_thread` (oopp-ex07 simplerunner.py)

- Funkcija **`start_new_thread`**, startuje novu nit za prosleđenu callback funkciju

```
import _thread, time
def handler(i):
    print(f"Hello from thread: {i}")

for i in range(10):
    _thread.start_new_thread(handler, (i,))

time.sleep(1)
```

Na izlazu je interesantan
rezultat

```
Hello from thread: 0Hello from thread: 1
Hello from thread: 3Hello from thread: 4
Hello from thread: 6
Hello from thread: 9

Hello from thread: 5
Hello from thread: 8Hello from thread: 2

Hello from thread: 7
```


Identifikacija niti

- Svaka kreirana nit dobija svoj identifikator koji vraća funkcija `start_new_thread`
- Ovaj identifikator možemo čuvati kako bi naknadno pristupili niti
- Takođe i sama nit može dobiti svoj identifikator

```
import _thread, time
def handler():
    id = _thread.get_ident()
    print(f"Hello from thread: {id}")

for i in range(10):
    id = _thread.start_new_thread(handler, ())
    print(f"Created thread with id: {id}")

time.sleep(1)
```



```
Created thread with id: 123145504641024
Created thread with id: 123145499385856
Created thread with id: 123145509896192
Created thread with id: 123145515151360
Created thread with id: 123145520406528
Created thread with id: 123145525661696
Created thread with id: 123145530916864
Created thread with id: 123145536172032
Created thread with id: 123145541427200
Hello from thread: 123145515151360
Hello from thread: 123145499385856
Hello from thread: 123145504641024
Hello from thread: 123145520406528
Hello from thread: 123145509896192
Hello from thread: 123145525661696
Hello from thread: 123145530916864
Hello from thread: 123145536172032
Hello from thread: 123145541427200
```

Upotreba modula threading

- Postoje dva načina na koje je moguće određeni kod izvršiti u zasebnoj niti pomoću modula threading. To su:
 - Umetanje callback funkcije u instancu klase **Thread**
 - Nasleđivanje klase **Thread** i implementacijom funkcije `run`
- Svaki od ova dva načina rezultiraće postojanjem metode ***run*** u klasi, u koju moramo smestiti logiku niti. Ovaj metod je prva tačka koja će se izvršiti nakon aktivacije niti i odatle možemo aktivirati ostalu funkcionalnost

Umetanje callback funkcije u klasu Thread (oopp-ex07 cbfinject.py)

- Klasa koja sledi je klasa koju će nit startovati. Ali samo ukoliko se to eksplicitno naglasi. Samo instanciranje klase koja sadrži callback funkciju, ne garantuje da će ona biti tretirana kao nit

```
def cbf():  
    id = threading.get_ident()  
    print(f"Hello from thread {id}")  
  
t = threading.Thread(None, cbf)  
  
for i in range(10):  
    t = threading.Thread(None, cbf)  
    t.start()
```

Ne startuje nit



Startuje nit



Umetanje callback funkcije u Thread klasu (oopp-ex07 threadinheritance.py)

- Osim implementacije runnable interfejsa, niti se mogu kreirati i nasleđivanjem klase **Thread**
- Kada je nit kreirana na ovaj način, ne mora imati omotač (klasu Thread), već se može upotrebljavati direktno

```
import threading

class MyThread(threading.Thread):
    def run(self):
        print("Thread is running")

mt = MyThread()
mt.start()
```

- Metod run nije neophodno implementirati, ali tada nasleđivanje nema naročit smisao

sleep funkcija

- Funkcija sleep, modula time zaustavlja nit na određeni period određen milisekundama (ili nanosekundama)
- Dok je nit zaustavljena, ostale niti mogu se izvršavati nesmetano

```
time.sleep(10)
```


Zaustavljanje niti

(oopp-ex07 closingthreads.py)

- Ukoliko aktiviramo metod `_stop` nad objektom koji se izvršava kao nit, nit će biti prekinuta trenutno
- Brutalno zaustavljanje niti metodom **`_stop`** čini da resursi koje je nit držala ne budu adekvatno raspušteni i time postanu nedostupni ostalim nitima. Umesto toga, preporučuje se kontrolni resurs, promenljiva, koju će nit stalno proveravati i na osnovu toga korigovati svoj tok
- Kada se metod `run` izvrši do kraja nit će automatski biti zaustavljena

```
mt = MyThread()
mt.start()
time.sleep(2)
try:
    mt._stop()
except:
    print("Hey, wake up!")
```

Ne bi trebalo

```
class MyThread(threading.Thread):
    active = False
    def stop_thread(self):
        self.active = False
    def run(self):
        self.active = True
        print("I am going to sleep 10 seconds")
        for i in range(10000):
            time.sleep(0.001)
            if not self.active:
                return
        print("Thread is running")

mt = MyThread()
mt.start()
time.sleep(2)
print("Hey, wake up!")
mt.stop_thread()
```

Kontrola toka niti

(oopp-ex07 vehiclethread.py)

- Obzirom da se često paralelno izvršava nit koja ima istu strukturu, često ćemo hteti da kontrolišemo tok te niti pomoću njenih unutrašnjih stanja
- Na primer, recimo da hoćemo da imamo nit kojom je predstavljeno vozilo. Vozilo može biti kamion i automobil.
U zavisnosti od toga koje je vozilo u pitanju, drugačija je i potrošnja goriva
- Tada možemo kroz konstruktor klase postaviti parametre koji će odrediti ponašanje niti

```
class Vehicle(threading.Thread):  
    def __init__(self, type):  
        super().__init__()  
        self.type = type  
        self.fuel = 10  
        if type == "truck":  
            self.consumption = .4  
            self.speed = 1.5  
        else:  
            self.consumption = .2  
            self.speed = 1
```

Vežba

(oopp-ex07 snailrun.py)

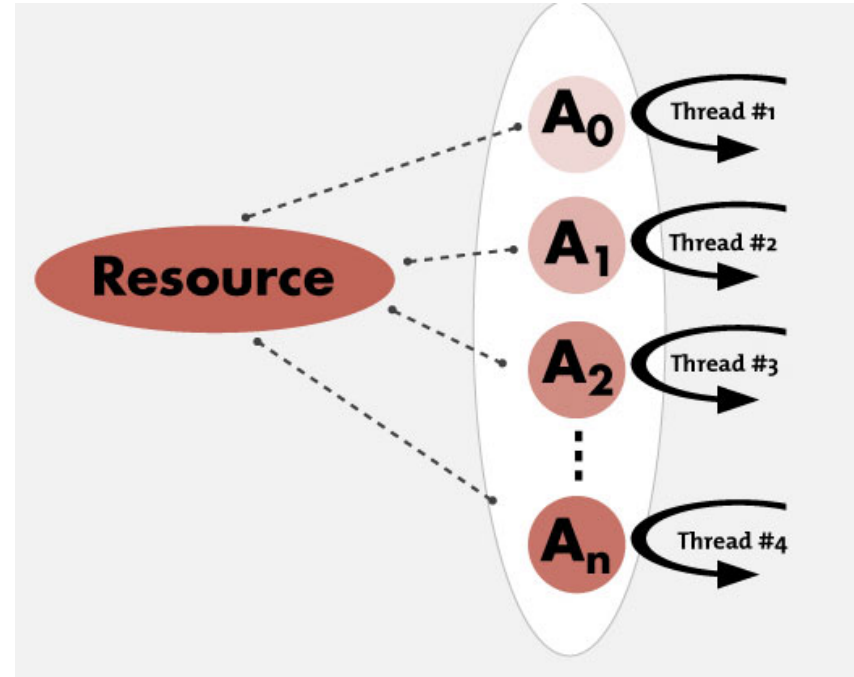
- Potrebno je napraviti program trka puževa
- Svaki puž predstavlja jednu Runnable klasu
- Run metod svake runnable klase sadrži petlju koja se izvršava deset puta i u svakoj iteraciji sadrži pauzu u slučajno izabranom intervalu od tri sekunde
- Svaki „puž“ (Runnable) ima svoju traku (kolonu) u kojoj se prikazuje do kog je broja trenutno došao
- Kada petlja u run metodi dođe do 0, ispisuje se na izlazu da je puž na toj niti došao do cilja
- Pomoć:
- Postavljanje stringa na određenu poziciju

```
print("1".rjust(5))  
print("2".rjust(10))
```

```
      3
          4  4
              1
                  2
                      3
                          4
                              3
                                  4
                                      1
                                          0
Snail #17 finished
```

Konkurencija i sinhronizacija niti

- U prethodnim primerima, niti su se izvršavale linearno, i koristile su isključivo resurse koji se nalaze unutar njih
- U praksi, niti će komunicirati sa resursima koji se nalaze izvan njih, i tada će se pojaviti dva nova problema o kojima se mora voditi računa
 - Dve niti pristupaju istim podacima / resursima
 - Jedan podatak je drugačije interpretiran od strane jedne i druge niti
- Ovi problemi rešavaju se sinhronizacijom



Global Interpreter Lock

- Različite platforme imaju ugrađene sisteme za praćenje upotrebe resursa - monitore
- Sistem u okviru monitor mehanizma koji omogućava konkurentno rukovanje resursima naziva se **Mutual Exclusion**, odnosno **Mutex**. Mutex je sistem koji onemogućava drugim nitima pristup resursu, sve dok određena nit rukuje njime. Kada nit završi proces nad resursom, Mutex dozvoljava pristup sledećoj niti koja se nalazi na listi čekanja pristup resursu
- Python ima poseban mehanizam koji se bavi samo problematikom konkurencije i taj mehanizam se naziva **Global Interpreter Lock (GIL)**.




Thread 1	running	waiting	running	waiting
Thread 2	waiting	running	waiting	running

Spajanje niti - join

- Metod **join** pozvan nad niti, blokira dalje izvršavanje programa, sve dok se nit ne izvrši

```
class MyThread(threading.Thread):  
    def run(self):  
        global numbers  
        for i in range(10):  
            print(i,end="")  
            time.sleep(0.5)  
mt1 = MyThread()  
mt2 = MyThread()  
mt1.start()  
mt1.join()  
print()  
mt2.start()
```



0123456789
0123456789

Spajanje niti - join

- Ukoliko se pozove sa parametrom, metod join prekida blokadu i nastavlja da se izvršava paralelno

```
class MyThread(threading.Thread):  
    def run(self):  
        global numbers  
        for i in range(10):  
            print(i,end="")  
            time.sleep(0.5)  
  
mt1 = MyThread()  
mt2 = MyThread()  
mt1.start()  
mt1.join(2)  
print()  
mt2.start()
```



Ovo traje
dve sekunde

```
01234  
0  
51627384956789
```

Od druge
sekunde

Zaključavanje resursa i sinhronizacija niti

- Osim GIL-a, Python omogućava ručnu sinhronizaciju niti, pomoću lock objekata
- Lock objekti su objekti koji se mogu zaključati i odključati (metode **acquire** i **release**)
- Ukoliko je lock objekat zaključan, nit čeka na njegovo otključavanje kako bi nastavila sa izvršavanjem
- Lock objekat se kreira instanciranjem klase **Lock**



Zaključavanje resursa i sinhronizacija niti

```
locker = threading.Lock()

class MyThread(threading.Thread):
    def run(self):
        locker.acquire()
        for i in range(10):
            print(i, end="")
            time.sleep(0.5)
        locker.release()

mt1 = MyThread()
mt2 = MyThread()
mt1.start()
mt2.start()
```



01234567890123456789



Komunikacija između niti

- Niti možemo i eksplicitno naglasiti da sačeka sve dok ne dobije naredbu da nastavi dalje. U tu svrhu koristimo klasu `Condition` i njene metode ***wait***, ***notify*** i ***notifyAll***
- Klasu `Condition` instanciramo sa prosleđenim **Lock** objektom kao parametrom ili bez njega (kada će klasa sama generisati ekskluzivni Lock objekat, instancu klase **RLock**)

```
locker = threading.Condition(threading.Lock())
```

Zadatak - Raskrsnica

- Kreirati program raskrsnica
- Program ima dve ulice koje se ukrštaju
- Automobili dolaze iz svakog pravca
- Potrebno je kreirati sistem semafora za raskrsnicu, tako da ne dođe do sudara ukoliko automobili poštuju saobraćajna pravila



Zadatak

- Kreirati program koji startuje pet niti. Svaka nit bira slučajan broj od 1 do 10. Nakon što sve niti završe sabiraju se sve generisane vrednosti

Executor

- Executor je alat koji posredno generiše i startuje niti
- Obzirom da se niti keširaju, ne postoji problem sa memorijom
- U slučaju prekoračivanja veličine keša, čeka se oslobađanje prostora

```
import concurrent.futures.thread as tr
import time

def f():
    print("Hello")
    time.sleep(5)
    print("World")

executor = tr.ThreadPoolExecutor(2)
executor.submit(f)
executor.submit(f)
```