به نام خدا

گزارش پروژه پردازش موازی

رضا نقدی

مقدمه:

ایده کلی انجام پروژه بخش بندی آن به سه دسته کلی: یک رابط کاربری ساده، کدهای هر تمرین (به صورت background processes)، اتصال و فرم دهی آها با استفاده از FastAPI می باشد بدین صورت که هر تمرین از طریق یک تابع در FastAPI یک URI دریافت می کند و پارامتر های ورودی آن تعیین شده و در نهایت خروجی هر کد دریافت و به رابط گرافیکی جهت نمایش ارسال می شود و در نهایت هم پروژه داکرایز شده است.

رابط گرافیکی:

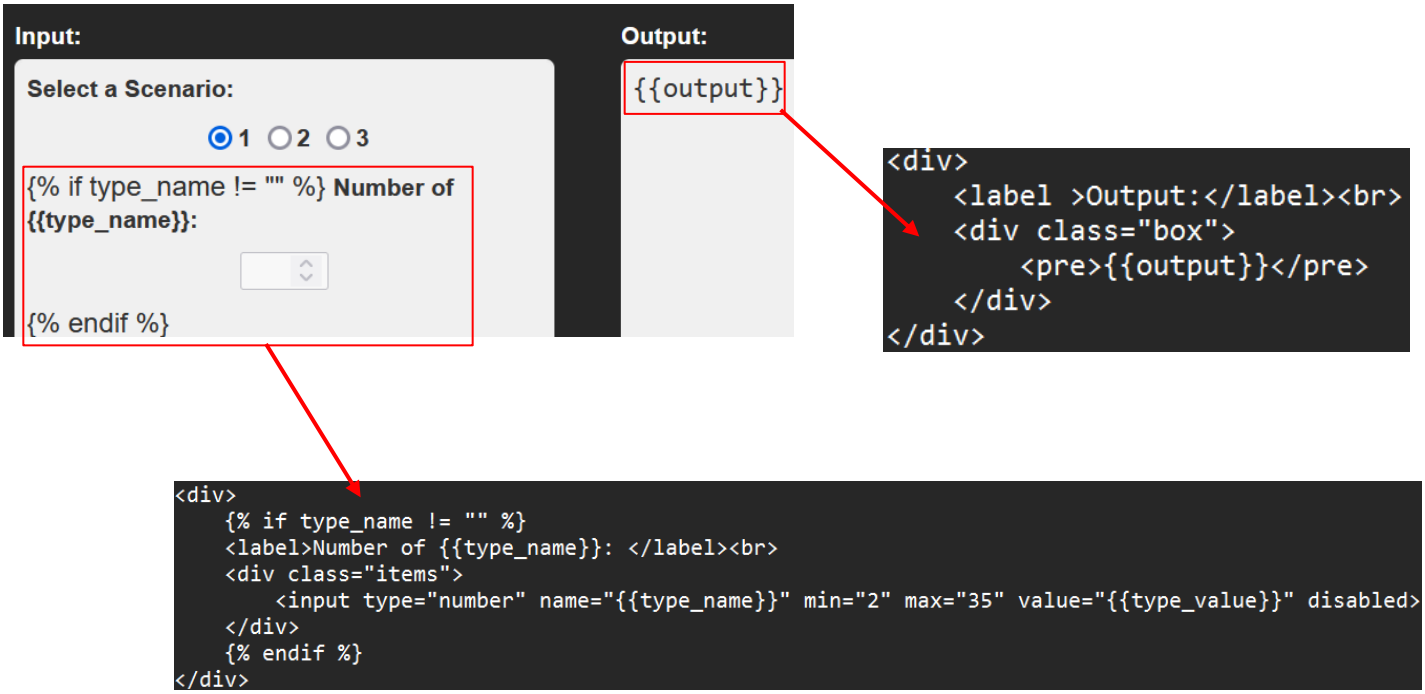
برای سهولت ارسال پارامترهای ورودی FastAPI که تماما به صورت متد GET هستند (استفاده از این متد به این دلیل بوده که راحت ترین نوع ارسال پارامتر است و در لینک ارسالی قرار می گیرد) و همچنین بازگرداندن خروجی که به صورت Json می باشد و سپس بخش اصلی آن به صورت Text در آمده از رابط کاربری ای به فرم HTML استفاده شده است و برای تمامی تمرینات از یک HTML فایل داینامیک که با شرط هایی داخل ساختار HTML، قابلیت ایجاد اندک تغییراتی را برای سازگار شدن با تمرینات گوناگون ایجاد گشته است.

ساختار دقیق تر رابط کاربری ما شامل یک فایل Menu و Showcase می باشد در بخش ابتدایی اجرای برنامه منو اجرا می شود که در آن نام هر تمرین به نمایش در می آید و هر کدام پارامتری را تحت عنوان exercise_tag به FastAPI می فرستد تا تمرین مربوطه که نحوه اجرای دقیق تر آن در ادامه خواهد آمد در فایل Showcase.HTML به نمایش در آید.

```
name="exercies_tag" value="s2e1"
name="exercies_tag" value="s2e2"
name="exercies_tag" value="s2e3"
name="exercies_tag" value="s2e4"
```

ساختار Showcase همانطور که اشاره شد به صورت داینامیک طراحی شده، برای هر تمرین ۴ بخش ثابت تصویر، ورودی ها، خروجی و توضیحات را داریم اما بخش های متغیر ما نام موجود در هدر، آدرس تصویر مربوط به هر تمرین، تعداد پارامتر های ورودی که برای هر تمرین پارامتر سناریو ثابت و یک بخش آپشنال تحت عنوان تعدادی از یک نوع خاص داریم و در نهایت خروجی و توضیحات که

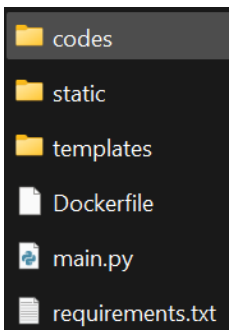
هر کدام از این موارد با یک متغیر داخل ساختار HTML مقدار دهی می شوند و جهت تعیین پارامتر نیز از نام استفاده می کنیم که در صورت تهی بودن نام آن پارامتر وجود ندارد، تمامی این موارد توسط هر تابع FastAPI بسته به هر تمرین به فایل showcase یا همان ویترین نمایش ما ارسال می شوند.



تصویر بالا نمونه ای از تعیین نوع داینامیک برای پارامتر دلخواه و خروجی تمرین می باشد و سایر موترد نیز به همین شکل تعریف شده اند.

FastAPI:

ساختار دهی FastAPI به گونه ای انجام شده که برای هر تمرین فرم کلی ای را در بر داشته باشد، پس هر تمرین موجود دارای شکل و فرم یکسانی را با تمرین های دیگر خواهد داشت، در ساختار فایل های پروژه ما فایل main.py که فایل اصلی برای اجرای پروژه است و کد های FastAPI را در بر می گیرد داریم و اما طبق روال عادی کارکرد FastAPI یا هر مورد دیگری بخش های دیگر نیز باید در کنار فایل اصلی قرار بگیرند تا بتوان از طریق FastAPI با آنها ارتباط برقرار کرد و این بخش ها شامل فولدر codes جهت قرار گیری فایل پایتون هر تمرین، فولدر static که شامل اجزا ثابت و لود شونده فایل های HTML مانند فایل های css و تصاویر می شود و در نهایت فولدر templates که فایل های HTML را در بر می گیرد.



در فایل main دو بخش اصلی جهت ارتباط با منو و showcase را داریم که showcase برای هر تابع ایجاد شده در هر تمرین ثابت و فرم اجرای تغریبا یکسانی را دارد.

```python
@app.get("/")
async def index(exercies_tag: str = None):
    if exercies_tag == None:
        return FileResponse("templates/menu.html")
    else:
        return RedirectResponse(f"/{exercies_tag}", status_code=303)
```

تابع تعریف شده برای اجرای منو برنامه و بارگیری تمرین مورد نظر تحت عنوان index و با URI ریشه یا به عبارتی "/" تعریف شده یعنی اگر ما آیپی یا دامنه ست شده را جست و جو کنیم ابتدا با منو مواجه می شویم در اینجا همانطور که گفته شده پارامتر exercires_tag را داریم که در ابتدا مقداری ندارد پس با برسی شرط صفحه منو بارگیری می شود FileResponse() و مقدار تمرین انتخابی با قرار گیری در URI مجدد به تابع index باز میگردد با برسی شرط این بار صفحه منو لود نمی شود و با جایگزاری مقدار مربوط به تگ تمرین به URI دیگری که تابع یک تمرین را در بر می گیرد ریدایرکت می شویم.

```python
@app.get("/s1e1")
async def Exercise_S1_E2(request: Request, scenario: int = None, n_threads: int = 10, Json: int = 0):
    output = "There is nothing to show !!!"
    if scenario != None:
        result = subprocess.run(['python', 'codes/s1e1.py', str(scenario), str(n_threads)], capture_output=True, text=True)
        output = json.dumps(result.stdout)
        output = eval(output)
    if Json:
        return result
    return templates.TemplateResponse("showcase.html",{"request": request, "scenario": scenario, "type_name": "threads", "type_value": n_threads, "output": output, "img": "s1e1.png"})
```

FastAPI Parameters

Excersice Parameters

HTML Parameters

هر تمرین دارای نام تابع و URI مختص خود اما با ساختار تقریبا مشابه می باشد بدین گونه که پارامتر های ورودی مشخص هستند و در صورتی که ابتدا سناریویی تعیین نشده باشد صفحه showcase.HTML به صورت خام اجرا می شود، یعنی تصویر تمرین و پارامتر ها و نام را نمایش می دهد و بخش توصیحات و خروجی مورد پیشرفرضی را قرار می دهد.

با انتخاب پارامتر ها و ارسال آنها مجدد صفحه لود می شود با این تفاوت که اینبار با برسی شرط کد مربوطه به تمرین تحت یک پراسس فرزند با subprocess با آرگومان های مد نظر اجرا شده و خروجی در result که به صورت JSON می باشد باز میگردد که قابلیت نمایش خام JSON و نمایش متن خروجی با تعیین پارامتر مربوط به آن را داریم.

تمرینات:

برای اجرای ۳ سناریوی متفاوت در هر تمرین یک کد واحد را با شرط های گوناگون پیاده سازی کرده ایم که هر شرط موجب اجر یا عدم اجرای دستوری برای هر سناریو خواهد شد، در ادامه سناریوی هر تمرین جداگانه و کامل مورد برسی قرار می گیرد:

- تمرین s1e1
  - سناریو ۱:

در این سناریو وجود join() باعث می شود که ترد اجرایی ابتدا تکمیل شود و سپس ترد بعدی به اجرا برود پس با وجود join() بلا فاصله پس از start() ترتیب اجرای ترد ها را خواهیم داشت.

  - سناریو ۲:

در صورت عدم استفاده از join() نحو دیگر اجرای ترتیبی تردها به این صورت می باشد که از یه وقفه زمانی به اندازه حداقل مدت زمان مورد نیاز برای اجرای ترد های فعلی در حال اجرا بهره ببریم که در اینجا ۰.۳ ثانیه صبر می کنیم و سپس ترد بعدی start() می شود.

  - سناریو ۳:

نحوه دیگر تضمین ترتیب با وقفه زمانی به این گونه است که از یک زمان افزایشی استفاده کرد، با استفاده از شماره هر ترد و ضرب آن در یک زمان ثابت زمان های از مضرب پایه انتخابی خواهیم داشت که همزمان با هم اجرا شده و در ترد های با شماره کمتر زمان انتظار زودتر به اتمام می رسد.

```python
from threading import Thread as T
import time
import sys

def func(num: int = 10)-> str:
    if scenario == 3:
        time.sleep(0.2 * num)
    print(f"Function called by thread {num}")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_threads = int(sys.argv[2])

    for i in range(n_threads):
        t = T(target=func, args=(i,))
        t.start()
        if scenario == 1:
            t.join()
        elif scenario == 2:
            time.sleep(0.3)
```

- تمرین s1e2
  - سناریو ۱:

در این سناریو هر ترد تابع مربوط به خودش را فرا می خواهد، نحوه چاپ ورود به تابع به هر ترتیبی می تواند صورت بگیرد اما معمولا چون در حد چند صدم ثانیه عمل start() برای هر ترد تا ترد بعدی زمان می برد ترتیب اجرا را خواهیم داشت و برای ترتیب اجرا در خروجی از وقفه های زمانی تصاعدی بهره گرفته ایم.

  - سناریو ۲:

در این سناریو ما اجرای دستور join() بعد از هر start() را خواهیم داشت که اجرای کامل هر ترد پیش از ارسال ترد بعدی برای اجرا را خواهیم داشت.

  - سناریو ۳:

در این پیاده سازی ما از زمان های انتظار نزولی بهره گرفته ایم که با توجه به اجرای با فاصله هر ترد در حد چند صدم ثانیه اجرای معکوس ترد ها موقع ورود و هنگام خروج را خواهیم داشت اما این امر تضمین شده نمی باشد.

```python
from threading import Thread as T
import time, sys, threading

def FA():
    if scenario == 3:
        time.sleep(0.3)
    print(f"Starting thread {threading.currentThread().getName()}")
    time.sleep(0.3)
    if scenario == 1:
        time.sleep(0.1)
    print(f"Exiting from thread {threading.currentThread().getName()}")

def FB():
    if scenario == 3:
        time.sleep(0.2)
    print(f"Starting function {threading.currentThread().getName()}")
    time.sleep(0.3)
    if scenario == 1:
        time.sleep(0.2)
    print(f"Exiting from function {threading.currentThread().getName()}")

def FC():
    if scenario == 3:
        time.sleep(0.1)
    print(f"Starting function {threading.currentThread().getName()}")
    time.sleep(0.3)
    if scenario == 1:
        time.sleep(0.3)
    print(f"Exiting from function {threading.currentThread().getName()}")

if __name__ == "__main__":
    scenario = int(sys.argv[1])

    A = T(target=FA, name= "A")
    B = T(target=FB, name= "B")
    C = T(target=FC, name= "C")
    A.start()
    if scenario == 2:
        A.join()
    B.start()
    if scenario == 2:
        B.join()
    C.start()
    if scenario == 2:
        C.join()
```

- تمرین s1e3
  - سناریو ۱:

    در این سناریو به دلیل اجرای با فاصله در حد صدم ثانیه هر ترد ورود به ترتیب به متد های کلاس را خواهیم داشت و اما به هنگام خروج به دلیل وجود وقفه رندوم ترتیب خاصی وجود نخواهد داشت و در نهایت وجود join() برای همه ترد ها موجب چاپ زمان اجرا در آخرین مرحله می شود.
  - سناریو ۲:

    در این سناریو به دلیل عدم وجود وقفه و join() نحوه اجرا و خروج از هر ترد کاملا بسته به منابع سیستم دارد و ترتیب های ورود و خروج کاملا به هم ریخته می باشند.
  - سناریو ۳:

    این سناریو با قرار دادن وقفه ۱ هزارم ثانیه ای نشان می دهد که سرعت انجام یک دستور در ترد در حدود هزارم ثانیه می باشد چرا که این وقفه موجب اجرای کامل ورود ها و سپس اجرای کامل خروج ها شده است و اگر آن را کاهش دهیم مانند سناریو قبل درهم ریختگی را مجددا خواهیم داشت

```python
from threading import Thread as T
from random import randint
import time, sys, os

class SubClass(T):
    def __init__(self, name, duration):
        T.__init__(self)
        self.name = name
        self.duration = duration

    def run(self):
        print (f"{self.name} running, process ID {str(os.getpid())}")
        time.sleep(self.duration)
        print (f"{self.name} over")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_threads = int(sys.argv[2])

    start_time = time.time()

    threads = []
    for i in range(1, n_threads):
        if scenario == 1:
            duration = randint(1,3)
        elif scenario == 2:
            duration = 0
        elif scenario == 3:
            duration = 0.001
        thread = SubClass(f"Thread {i}", duration)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()

    print(f"\n-- {(time.time() - start_time)} seconds --\n")
```

- تمرین s1e4
  - سناریو ۱:

در این سناریو که در واقع مشابه تمرین قبل می باشد مشاهده می شود که با یک زمان ثابت نیم ثانیه ای خروجی قبلی که ترتیب چاپ ورود ها و سپس چاپ خروج ها بود را نداریم و ورود خروج هر ترد پشت سر هم آمده که علت وجود lock می باشد و چون کل متد ما ناحیه بحرانی در نظر گرفته شده تا خروج کامل از ترد سایر ترد ها به حالت انتظار در خواهند آمد.

  - سناریو ۲:

در این سناریو برای اثبات عملکرد lock زمان مابین چاپ ورود و خروج برداشته شده است و همانطور که می بینیم ترتیب اجرا با سناریو قبل یکسان خواهد بود.

  - سناریو ۳:

در اینجا چون کل اجرای یک ترد ما ناحیه بحرانی اطلاق شده است می توان به جای استفاده از لاک از جوین بهره گرفت و مشاهده کرد که خروجی مشابه سناریوهای قبلی می باشد.

```python
from threading import Thread as T
import time, sys, os, threading

class SubClass(T):
    def __init__(self, name, duration):
        T.__init__(self)
        self.name = name
        self.duration = duration

    def run(self):
        if scenario == 3:
            print (f"{self.name} running, process ID {str(os.getpid())}")
            time.sleep(self.duration)
            print (f"{self.name} over")
        else:
            with Lock:
                print (f"{self.name} running, process ID {str(os.getpid())}")
                time.sleep(self.duration)
                print (f"{self.name} over")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_threads = int(sys.argv[2])

    start_time = time.time()
    Lock = threading.Lock()

    threads = []
    for i in range(1, n_threads):
        if scenario != 2:
            duration = 0.5
        elif scenario == 2:
            duration = 0
        thread = SubClass(f"Thread {i}", duration)
        threads.append(thread)
        thread.start()
        if scenario == 3:
            thread.join()
    for i in threads:
        thread.join()

    print(f"\n--- {(time.time() - start_time)} seconds ---")
```

- تمرین s1e5
  - سناریو ۱:

در این تمرین نحوه کارکرد rlock به چشم می خورد که امکان دریافت چندباره توسط یک ترد را دارا می باشد و اما در این سناریو وجود وقفه ۰.۲ ثانیه ای موجب اجرای تقریبا یک در میان دو ترد می شود و در نهایت به دلیل بیشتر بودن تعداد اجرای افزایش در آخر فقط ترد افزودن اجرا می شود.

  - سناریو ۲:

در اینجا زمان وقفه برداشته شده و می بینیم که به دلیل عدم وجود وقفه ترتیب اجرا تقریبا اجرای کامل یک ترد و سپس دیگری می باشد

سناریو ۳:

برای اجرای کاملا به ترتیب این دو ترد از Join() استفاده شده است در این حالت می توان از rlock نیز استفاده نکرد چرا که در هر واحد زمانی تنها یک ترد به متغیر ناحیه بحرانی ما دسترسی خواهد داشت همچنین به دلیل اینکه عملیات های ما تنها افزایش و کاهش هستند دسترسی همزمان ترد ها مشکلی اجاد نمیکند فقط ترتیب کاهش ها و افزایش ها است که ممکن است مسئله ساز شود.

```python
import time, sys, random, threading

class execution():
    def __init__(self):
        self.rlock = threading.RLock()
        self.total_items = 0

    def execute(self, value):
        with self.rlock:
            self.total_items += value

    def add(self):
        with self.rlock:
            self.execute(1)

    def remove(self):
        with self.rlock:
            self.execute(-1)

def adder(exec, items):
    print(f"{items} items to ADD\n")
    while items:
        exec.add()
        items -= 1
        if scenario != 2:
            time.sleep(0.2)
        print(f"+ One item ADDED, {items} remained.")

def remover(exec, items):
    print(f"\n{items} items to REMOVE\n")
    while items:
        exec.remove()
        items -= 1
        if scenario != 2:
            time.sleep(0.2)
        print(f"- One item REMOVED, {items} remained.")


if __name__ == "__main__":
    scenario = int(sys.argv[1])

    exec = execution()
    adder_thread = threading.Thread(target=adder, args=(exec, random.randint(10,15)))
    remover_thread = threading.Thread(target=remover, args=(exec, random.randint(5,10)))
    adder_thread.start()
    if scenario == 3:
        adder_thread.join()
    remover_thread.start()
```

- تمرین s1e6
  - سناریو ۱:

در اینجا تعداد n جفت ترد به صورت همزمان اجرا شده اند که ترد ابتدایی مصرف کننده است و ترد ثانویه که مدت زمانی را برای اجرا نیاز دارد تولید کننده پس چون ابتدا آیتمی وجود ندارد تمامی ترد های مصرف کننده به حالت تعلیق در آمده و با تولید یک آیتم سمافور افزایش می یابد و یکی از ترد های معلق جهت اجرا آزاد می شود و به همین ترتیب اجرای یک در میان تولید و مصرف را داریم، در صورتی که تولیدی صورت نگیرد مصرف کننده تا ابد در حالت تعلیق می ماند.

  - سناریو ۲:

در این حالت تولید کننده به زمان وابسته نمی باشد و میزان حالت هایی که کمبود آیتم داریم و نیاز است تا ترد مصرف کننده ای تعلیق شود کمتر است.

  - سناریو ۳:

در این سناریو تولید کننده غیر وابسته به زمان را ابتدا اجرا کرده و سپس مضرف کننده ای را فرا میخوانیم اینگونه دیگر زمان انتظار و تعلیقی را نخواهیم داشت.

```python
import time, sys, random, threading

def consumer():
 global item
 if not semaphore._value:
    print("Consumer is waiting ...")
 semaphore.acquire()
 print(f"Consumer notify: item number {item}")

def producer():
 global item
 if scenario == 1:
    time.sleep(0.2)
 item = random.randint(0, 1000)
 print(f"Producer notify: item number {item}")
 semaphore.release()

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_threads = int(sys.argv[2])

    semaphore = threading.Semaphore(0)
    item = 0
    for i in range(n_threads):
        cons = threading.Thread(target=consumer)
        prod = threading.Thread(target=producer)
        if scenario != 3:
            cons.start()
        prod.start()
        if scenario == 3:
            cons.start()
```

- تمرین s1e7
  - سناریو ۱:

    در این تمرین ما به مفهوم مانع اشاره می کنیم که با برخورد ۳ ترد با آن اجازه ادامه داده می شود
    اما در این سناریو نحوه اجرای زمان به این صورت است که رندوم یک زمان به هر ترد اختصاص داده می شود اما زمان تمامی این تردها با هم شروع می شود یعنی این زمان مدت زمان کل ترد از ابتدا تا انتها می باشد و در اینجا هر ترد ممکن است زودتر به از سایرین به پایان برسد
  - سناریو ۲:

    در این سناریو تردی که ابتدا اجرا شده با سپری کردن زمان خود نیز زود تر به پایان می رسد و این زمان ها همچنان رندوم می باشند
  - سناریو ۳:

    ترد ها با فاصله زمانی ۱ ثانیه از یکدیگر به پایان می رسند یعنی زمان بندی به صورت ۱، ۲، ۳ و ... می باشد.

```python
from random import randrange
from threading import Barrier, Thread
from time import ctime, sleep
import sys

def runner(num):
    global runners, finish_line
    if scenario == 1:
        sleep(randrange(1, 4))
    elif scenario == 2:
        sleep((num + 1) * randrange(1, 4))
    elif scenario == 3:
        sleep(num + 1)
    print(f"'{runners.pop()}' reached the barrier:\n--> {ctime()} \n")
    finish_line.wait()

if __name__ == "__main__":
    scenario = int(sys.argv[1])

    runners = ['Reza', 'Ali', 'Hassan']
    finish_line = Barrier(len(runners))

    threads = []
    print('START RACE!!!!\n')
    for i in range(len(runners)):
        threads.append(Thread(target=runner, args=(i, )))
        threads[-1].start()
    for thread in threads:
        thread.join()
    print('RACE OVER!')
```

❖ در تمرینات سری ۲ به دلیل اینکه هر پراسس حافظه مجزا دارد باید مقادیر متغیرها را با استفاده از pipeline، queue، و پارامترهای ارسالی تبادل کنند

- تمرین s2e1
  - سناریو ۱:

در این سناریو شماره هر پراسس به تابع فراخوانی کننده اش ارسال و یک حلقه داخل تابع از صفر تا شماره پراسس چاپ می کند و دستور Join() تضمین کننده اجرای ترتیبی است

  - سناریو ۲:

در این سناریو به دلیل عدم وجود join() تمامی پراسس ها همزمان اجرا شده پیام وذود را چاپ و ۰.۲ ثانیه انتظار می کشند سپس حلقه تکرار خود را اجرا میکنند که ترتیب اجرایی نخواهیم داشت

  - سناریو ۳

جهت برطرف سازی ضعف ایجاد شده در سناریو قبل می توان از سمافور بهره گرفت اما باید توجه داشت که سمافور به عنوان پارامتر به هر پراسس ارسال شود و حلقه را به عنوان ناحیه بحرانی در نظر گرفت تا چاپ های هر پراسس با دیگری ادغام نشود.

```python
from multiprocessing import Process as p, Semaphore
import sys, time

def func(i, scenario, s):
    print (f"calling func from process: {i}")
    if scenario != 1:
        time.sleep(0.2)
    with s:
        if scenario != 3:
            s.release()
        for j in range (0,i):
            print(f"func output: {j}")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_process = int(sys.argv[2])

    s = Semaphore(1)
    for i in range(n_process):
        process = p(target= func, args=(i, scenario, s))
        process.start()
        if scenario != 2:
            process.join()
```

- تمرین s2e2
  - سناریو ۱:

در این سناریو هر دو پراسس اجرا شده پیغام ورود را چاپ و سپس خارج می شوند، مشاهده می شود که یک پراسس دارای نام تعیین شده و دیگری دارای یک نام پیشفرض است

  - سناریو ۲:

در اینجا ترتیب ورود و خروج ها برای اجرای کامل یک پراسس را با استفاده از join() مشاهده می کنیم

  - سناریو ۳:

در این سناریو ابتدا تابع با نام تعریف شده استارت شده است اما نام آن را ابتدا نمی بینیم چرا که پراسس فرزند پس از اجرا خود پراسسی را تحت نام insider فراخوانده در این مدت زمان پراسس با نام پیشفرض اجرا شده و پس از وارد شدن به پراسس insider دستور چاپ پراسس والد آن را داریم insider نیز پراسسی را فراخوانده که آرگومان آن None است و اگر مجدد آرگومان ۳ ارسال شود فرایند اجرای پراسس های فرزند بی انتها می شد

```python
from multiprocessing import Process as p
import time, sys, multiprocessing as mp

def func(scenario):
    if scenario == 3:
        p(name= "insider", target= func, args=(None, )).start()
    name = mp.current_process().name
    print (f"Starting process name = {name}\n")
    time.sleep(1)
    print (f"Exiting process name = {name}\n")

if __name__ == "__main__":
    scenario = int(sys.argv[1])

    process_with_name = p(name="myFunc process",target= func, args= (scenario, ))
    process_with_default_name = p(target= func, args=(None, ))

    process_with_name.start()
    if scenario == 2:
        process_with_name.join()
    process_with_default_name.start()
```

- تمرین s2e3
  - سناریو ۱:

در اینجا ۲ پراسس که یکی در حالت بکگراند و دیگری نرمال اجرا می شود داریم و پراسس اول اعداد زوج صفر تا نه و پراسس دوم اعداد فرد را چاپ می کند می بینیم که بکگراند پراسس تعاملی با خروجی ندارد

  - سناریو ۲:

در اینجا شرایط بکگراند بودن را برای پراسس ابتدایی برداشته و مشاهده می شود که تمامی اعداد صفر تا نه ما بین یکدیگر و به صورت تقریبا ترتیبی چاپ می شوند

  - سناریو ۳:

جهت درک بهتر تعاملی بودن و اینکه کدام خروجی مربوط به کدام پراسس است از join() استفاده می کنیم تا متوجه بشیم خروجی ای که در سناریو ابتدایی وجود نداشته است دقیقا چه مواردی بوده اند

```python
import multiprocessing
import time, sys

def func():
    name = multiprocessing.current_process().name
    print (f"Starting {name}\n")
    time.sleep(0.1)
    if name == 'background_process':
        for i in range(0, 10, 2):
            print(f"---> {i}\n")
            time.sleep(0.2)
    else:
        for i in range(1, 10, 2):
            print(f"---> {i}\n")
            time.sleep(0.2)
    time.sleep(0.1)
    print (f"Exiting {name}\n")

if __name__ == "__main__":
    scenario = int(sys.argv[1])

    background_process = multiprocessing.Process(name="background_process",target=func)
    if scenario != 2:
        background_process.daemon = True
    NO_background_process = multiprocessing.Process(name="NO_background_process",target=func)
    NO_background_process.daemon = False
    background_process.start()
    if scenario == 3:
        background_process.join()
    NO_background_process.start()
```

- تمرین s2e4
  - سناریو ۱:

در اینجا مشاهده می شود که یک پراسس ما پیش از اجرا حالت اجرای False هنگام اجرا True موقع Terminate به دلیل اینکه کمی این امر زمانبر است همچنان True اما چون پراسس متوقف شده خروجی تابع را نخواهیم دید و پس از این امر با دستور جوین False خواهد بود همچنین کد خروج ما منفی می باشد به این دلیل که سیگنالی خارجی موجب توقف پراسس شده است

  - سناریو ۲:

اینبار پیش از Terminate کردن پراسس کمی به اجرای پراسس زمان می دهیم تا مشاهده شود پراسس در حال اجرا و چاپ خروجی است سپس آن را Terminate می کنیم.

  - سناریو ۳:

گفته شد هنگام تغییر وضعیت پراسس کمی زمانبر است تا حالت درست آن به نمایش در آید پس اینبار مقداری وقفه زمانی پیش از چاپ وضعیت Terminated اضافه کردیم می بینیم که پس از آن وضعیت اجرا False است.

```python
import multiprocessing
import time, sys

def func():
    print ("Starting function\n")
    for i in range(0,5):
        print(f"--> {i}\n")
        time.sleep(0.2)
    print ("Finished function")

if __name__ == "__main__":
    scenario = int(sys.argv[1])

    p = multiprocessing.Process(target=func)
    print ("Process before execution:", p.is_alive())
    p.start()
    if scenario == 2:
        time.sleep(1)
    print ("Process running:", p.is_alive())
    p.terminate()
    if scenario == 3:
        time.sleep(1)
    print ("Process terminated:", p.is_alive())
    p.join()
    print ("Process joined:", p.is_alive())

    print ("Process exit code:", p.exitcode)
```

- تمرین s2e5

  - سناریو ۱:

در اینجا پراسس های اجرا شده یک شی از کلاس مالتی پراسسینگ هستند که هرکدام نام خود را دارا می باشند و بخاطر وجود join() ترتیب اجرای آنها قابل مشاهده است

  - سناریو ۲:

در اینجا join() برداشته شده است و ترتیب خروجی ها نیز تغییر امکان تغییر دارد

  - سناریو ۳:

در این سناریو یک در میان برای هر پراسس جوین را خواهیم داشت که ترتیب در پراسس های زوج را به ارمغان می آورد و امکان اجرای متغیر در پراسس های فرد را خواهیم داشت

```python
import multiprocessing, sys, time

class MyClass(multiprocessing.Process):
    def run(self):
        time.sleep(0.5)
        print (f"called run method by {self.name}")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_process = int(sys.argv[2])

    for i in range(n_process):
        process = MyClass()
        process.start()
        if scenario != 2:
            process.join()
        elif scenario == 3 and i%2 == 0:
            process.join()
```

- تمرین s2e6
  - سناریو ۱:

در اینجا پراسس های تولید کننده و مصرف کننده با هم شروع به اجرا می کنند و چون تولید کننده کمی زود تر شروع به اجرا کرده روند تولید را داریم و از این پس زمان تولید مجدد ۲ برابر کمتر از مصرف است پس تا انتها تولید و مصرف ترتیبی را خواهیم داشت

  - سناریو ۲:

در اینجا ابتدا یک آیتم تولید می شود و مصرف کننده نیز آن را مصرف می کند اما چون مدت زمان انتظار مصرف کننده کوتاه است پیش از تولید آیتم بعدی اجرا شده و به دلیل عدم وجود آیتم متوقف می شود و از این پس تنها تولید بدون مصرف خواهیم داشت

  - سناریو ۳:

در این سناریو تولید کننده زمان انتظار ندارد پس به محض اجرا تعداد آیتم های تعیین شده را تولید میکنید و سپس مصرف کننده می تواند به مصرف بپردازد

```python
import multiprocessing
import time, sys, random

class producer(multiprocessing.Process):
    def __init__(self, queue, n_products, scenario):
        multiprocessing.Process.__init__(self)
        self.queue = queue
        self.n_products = n_products
        self.scenario = scenario
    def run(self) :
        for _ in range(self.n_products):
            item = random.randint(0, 256)
            self.queue.put(item)
            if self.scenario != 3:
                time.sleep(0.2)
            print (f"Producer : {item} appended to Q")
            print (f"The size of Q is {self.queue.qsize()}\n")


class consumer(multiprocessing.Process):
    def __init__(self, queue, scenario):
        multiprocessing.Process.__init__(self)
        self.queue = queue
        self.scenario = scenario
    def run(self):
        while True:
            if (self.queue.empty()):
                print("The Q is empty")
                break
            else :
                if self.scenario == 1:
                    time.sleep(0.4)
                item = self.queue.get()
                print (f"Consumer : {item} popped from Q\n")

if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_products = int(sys.argv[2])

    queue = multiprocessing.Queue()
    producer(queue, n_products, scenario).start()
    consumer(queue, scenario).start()
```

- تمرین s1e7
  - سناریو ۱:

در اینجا ۲ پراسس ۱ و ۲ به دلیل اینکه با مانع اجرا می شوند تا هر دو به مانع برسند کمی زمان می برد پس اپتدا پراسس ۳ و ۴ را خواهیم داشت و سپس ۱ و ۲

  - سناریو ۲:

در اینجا مانع برداشته شده و امکان اجرا به هر ترتیبی امکان دارد و منطقا انتظار می رود پراسس ۱ اول اجرا شود اما چون در تابع اجرایی این پراسس یک شرط اضافه به نسبت پراسس ۳ و ۴ وجود دارد ابتدا براسس ۳ سپس ۱ اجرا می شود اما چون در این حین پراسس ۲ فرصت داشته ترتیب ۲ و ۴ به هر طریقی ممکن است

  - سناریو ۳:

در اینجا جوین مابین جفت پراسس های ۱،۲ و ۳،۴ داریم که ترتیب اجرایی را تنها به ۲ گروه تقسیم می کند ابتدا ۱ و ۲ اجرا می شوند و کاملا همزمان هستند پس ترتیبی ما بین ۱ و ۲ نیست سپس ۳ و ۴ که این دو نیز بدین شکل هستند.

```python
from datetime import datetime

def test_with_barrier(synchronizer, serializer, scenario):
    name = multiprocessing.current_process().name
    if scenario != 2:
        synchronizer.wait()
    now = time()
    with serializer:
        print(f"process {name}:\n{datetime.fromtimestamp(now)}\n")

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print(f"process {name}:\n{datetime.fromtimestamp(now)}\n")


if __name__ == "__main__":
    scenario = int(sys.argv[1])

    synchronizer = Barrier(2)
    serializer = Lock()
    p1= Process(name="p1 - test_with_barrier", target=test_with_barrier, args=(synchronizer,serializer, scenario))
    p1.start()
    p2= Process(name="p2 - test_with_barrier", target=test_with_barrier, args=(synchronizer,serializer, scenario))
    p2.start()
    if scenario == 3:
        p1.join()
        p2.join()
    Process(name="p3 - test_without_barrier", target=test_without_barrier).start()
    Process(name="p4 - test_without_barrier", target=test_without_barrier).start()
```

- تمرین s2e8
  - سناریو ۱:

در اینجا با تابع map به pool پردازنده ها مقادیر و تابع مد نظر را می فرستیم و خروجی را دریافت میکنیم و لیستی از مقادیر خروجی با همان ترتیب ورودی با اینکه یک وقفه رندوم ایجاد کرده ایم خواهیم داشت.

  - سناریو ۲:

در این سناریو برسی می شود تا علت ترتیب آیتم ها با وجود وقفه رندوم مشخص گردد و قبل مشاهده است که تابع مپ نتایج خروجی را به صورت تصادفی جمع آوری و در نهایت همه را تجمیع و باز گردانی می کند و اگر پیش از تجمیع نتایج نمایش داده شوند بدون ترتیب خواهند بود

  - سناریو ۳:

در این سناریو برسی می شود تاثیر join() بر حالت نهایی و ترتیب گونه خروجی مپ چگونه است، با برداشتن این دستور مشاهده می شود که تاثیری در ترتیب خرجی حتی با وقفه های نا منظم نخواهیم داشت و تنها تاثیر join() در اینجا موارد اجرایی پس از این دستور مپ می باشد.

```python
import multiprocessing, sys, time, random

def function_square(data):
    time.sleep(random.randrange(0,2))
    return data * data

def function_square_print(data):
    time.sleep(random.randrange(0,2))
    print("-->",data * data)


if __name__ == "__main__":
    scenario = int(sys.argv[1])
    n_inputs = int(sys.argv[2])

    inputs = list(range(0, n_inputs))
    pool = multiprocessing.Pool(processes=2)
    pool_outputs = pool.map(function_square, inputs)
    if scenario == 2:
        print ("Pool :\n")
        pool.map(function_square_print, inputs)
    pool.close()
    if scenario != 3:
        pool.join()

    if scenario  != 2:
        print ("Pool :\n")
        for output in pool_outputs:
            print(f"--> {output}")
```

Docker:

جهت داکرایز کردن پروژه به ۲ فایل Dockerfile و Requirements.txt نیاز داریم، کتابخانه های مورد نیاز جهت اجرا را در Requirements گرد آوری می کنیم.

```
FROM python:3.9-slim

WORKDIR /
COPY . /

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 80

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

- ابتدا یک ورژن از پایتون را جهت اجرای پروژه فرا میخوانیم
- سپس آدرس دایرکتوری پروژه به نسبت محل داکر فایل را تنظیم کرده
- تعیین می کنیم موارد در کانتینر با چه آدرسی ذخیره سازی شوند
- کتابخانه های پیش نیاز را با pip نصب می کنیم
- پورت مورد نظر جهت اجرای برنامه را جهت listen آماده میکنیم (پورت ۸۰ یا ۴۴۳)
- دستور اجرای فایل اصلی برنامه که با uvicorn انجام میگیرد و آدرس هاست و آیپی اجرایی را دریافت می کند.

جهت ساخت کانتینر و اجرای آن توسط داکر نیاز به دستورات زیر هنگامی که در مسیر جاری پروژه هستیم داریم:

ساخت کانتینر:

```
docker build -t <NAME> <PATH>
```

اجرای کانتینر:

```
docker run -d -p <LAN_PORT>:<WAN_PORT>
```

توقف اجرا:

```
docker stop <NAME>
```

حذف کانتینر:

```
docker rm <NAME>
```