پروژه شبیه سازی
ترم ۲-۱۴۰۱

اعضا:

فرهاد اسماعیل زاده ۹۹۱۰۵۲۲۶
امیررضا ابوطالبی ۹۹۱۰۵۱۹۷

گرفتن ورودی ها

ابتدا ورودی ها که شامل X، Y، Z، تعداد پردازه ها(count_tasks) و مدت زمان شبیه سازی(time_of_simulation) است، گرفته می شوند.

```
# GETTING INPUTS
X = int(input("X:    "))
Y = int(input("Y:    "))
Z = int(input("Z:    "))
count_tasks = int(input("count tasks:    "))
time_of_simulation = int(input("time of simulation:    "))
```

لایه اول (فرآیند ایجاد تسک)

کلاس task که شامل attribute های زیر می باشد:
Interval: زمان ورود بین دو تسک
Service_time: مدت زمان اجرای تسک
Priority:اولویت
Time_queue: مدت زمان حضور در صف

```
# LAYER ONE : CREATING TASK
class Task:
    def __init__(self, interval, service_time, priority):
        self.interval = interval
        self.service_time = service_time
        self.priority = priority
        self.time_queue = 0
```

کلاس queue که شامل attribute های زیر می باشد:
Name: نام صف
Quantom_time: زمان کوانتوم
Tasks: تسک های موجود در صف

```python
class Queue:
    def __init__(self, name, quantum_time):
        self.name = name
        self.quantum_time = quantum_time
        self.tasks = []
```

تابع JobCreator:
ابتدا زمان های بین هر تسک را با استفاده از کتابخانه numpy بوسیله فرایند پواسون با نرخ X تولید میکنیم. برای پیدا کردن حداقل زمان اجرای تسک، از الگوریتم partial sums استفاده میکنیم،بدین صورت که هر interval را با تمام interval های تولید شده قبل آن جمع می کنیم.
سپس زمان اجرای هر تسک را با استفاده از کتابخانه numpy بوسیله توزیع نمایی با میانگین Y تولید میکنیم.اگر هرکدام از زمان های اجرا ۰ بود، انقدر یک نمونه دیگر برای آن تولید میکنیم که دیگر ۰ نباشد.
بعد نوبت به اولویت می رسد که آن را نیز مانند قسمت های قبل نیز با استفاده از کتابخانه numpy و با متود choice تولید می کنیم.
حال تمام اطلاعات مورد نیاز برای ایجاد تسک را داریم.به ازای هر زمان بین ورود، زمان اجرا و اولویت یک instance از کلاس task ایجاد میکنیم و آن را در ارایه ای قرار می دهیم و در آخر ان ارایه را return میکنیم.
بعد از آن باید خروجی این تابع را در صف priority queue قرار دهیم که آن را ایجاد میکنیم.

```python
def JobCreator():
    intervals = numpy.random.poisson(X, count_tasks)
    intervals = [sum(intervals[:i]) for i in range(1, count_tasks + 1)]
    service_times = [int(x) for x in numpy.random.exponential(Y, count_tasks)]
    for i in range(count_tasks):
        while True:
            if service_times[i] == 0:
                service_times[i] = int(numpy.random.exponential(Y, 1))
            else:
                break
    priorities = numpy.random.choice(["Low", "Normal", "High"], count_tasks, True, [0.7, 0.2, 0.1])
    return [Task(intervals[i], service_times[i], priorities[i]) for i in range(count_tasks)]
```

لایه دوم (انتقال)

دیگر صف های موجود را ایجاد میکنیم.
متغیر time_period که مدت زمان بین صدا زدن تابع JobLoading است را مقداردهی میکنیم.
همچنین متغیر k که تعداد تسک های که میخواهیم هر دفعه از priority queue به صف های لایه دوم منتقل کنیم را مقدار دهی میکنیم.
برای اولویت بندی انتقال تسک ها از لایه اول به لایه دوم، تسک را ها بر اساس زمان بین ورودشان به صورت نزولی مرتب سازی میکنیم.

```python
# LAYER TWO : TRANSFERRING
FCFS = Queue("FCFS", math.inf)
Round_Robin_T1 = Queue("Round_Robin_T1", 6)
Round_Robin_T2 = Queue("Round_Robin_T2", 10)
time_period = 15
k = 5
priority_queue.tasks.sort(key=lambda x: x.interval, reverse=False)
```

تابع JobLoading:
اگر تعداد تسک های موجود در صف های لایه دوم از k کمتر بود،فرایند را شروع می کنیم.
به اندازه حداکثر k، هر تسکی را که به حداقل زمان شروعش رسیده ایم را از priority queue حذف میکنیم و اگر زمان اجرایش از quantum time صف Round-Robin-T1 کمتر بود، آن را به همین صف خواهیم داد. در غیر این صورت چک میکنیم که ایا از quantum time صف Round-Robin-T2 کمتر است یا خیر.اگر کمتر بود، آن را به صف Round-Robin-T2 و اگر کمتر نبود، به صف FCFS می دهیم.

```
def JobLoading(time):
    if len(FCFS.tasks) + len(Round_Robin_T1.tasks) + len(Round_Robin_T2.tasks) < k:
        counter = 0
        for task in priority_queue.tasks:
            if counter == k or task.interval > time:
                break
            if task.service_time <= Round_Robin_T1.quantum_time:
                Round_Robin_T1.tasks.append(task)
            elif task.service_time <= Round_Robin_T2.quantum_time:
                Round_Robin_T2.tasks.append(task)
            else:
                FCFS.tasks.append(task)
            priority_queue.tasks.remove(task)
            counter += 1
```

فرایند انجام تسک

کلاس DoneTask که در واقع همان کلاس task به اضافه زمان شروع پردازش تسک است.

```python
# RUNNING TASKS
class DoneTask:
    def __init__(self, task, time_start):
        self.task = task
        self.time_start = time_start
```

کلاس CPU که شامل attribute های زیر می باشد:
Assigned_task: تسکی که در حال حاضر پردازش می شود
Time_working: مدت زمانی که cpu مشغول بوده است
Time_start_task: زمان شروع تسکی که در حال پردازش است
Done_tasks: تسک های انجام شده
Length_queue: ارایه ای برای نگهداری تعداد تسک ها در صف های لایه دوم در ثانیه

```python
class CPU:
    def __init__(self):
        self.assigned_task = None
        self.time_working = 0
        self.time_start_task = 0
        self.done_tasks = []
        self.length_queue = []
```

از کلاس CPU یک instance می گیریم.

تابع assign_task_to_CPU:
اگر هیچ تسکی در هیچکدام از صف های لایه دوم وجود نداشته باشد، تسکی برای پردازش در حال حاضر موجود نیست.
در غیر اینصورت، با احتمالات داده شده یک صف را انتخاب میکنیم(انقدر این کار را انجام میدهیم تا بالاخره یک صفی انتخاب شود که خالی نباشد) و از ان اولین تسک را میگیریم و به cpu می دهیم و زمان اجرای تسک را ثبت میکنیم.

```python
def assign_task_to_CPU(current_time):
    if len(FCFS.tasks) == 0 and len(Round_Robin_T1.tasks) == 0 and len(Round_Robin_T2.tasks) == 0:
        return
    while True:
        chosen = numpy.random.choice(["FCFS", "Round_Robin_T1", "Round_Robin_T2"], 1, True, [0.1, 0.8, 0.1])
        if chosen == "FCFS":
            chosen_queue = FCFS
        elif chosen == "Round_Robin_T1":
            chosen_queue = Round_Robin_T1
        else:
            chosen_queue = Round_Robin_T2
        if len(chosen_queue.tasks) != 0:
            break
    cpu.assigned_task = chosen_queue.tasks[0]
    chosen_queue.tasks.pop(0)
    cpu.time_start_task = current_time
```

سپس مقدار timeout را با استفاده از کتابخانه numpy بوسیله توزیع نمایی با میانگین Z تولید میکنیم.

حال یک حلقه برای شبیه سازی می زنیم که مقدار آن تا مدت زمان شبیه سازی(time_of_simlation) میرود.
در این حلقه اگر زمان مضربی از time_period باشد، تابع JobLoading را صدا میزنیم.
سپس مدت زمان حضور تسک های هر صف را اپدیت میکنیم.
اگر cpu بیکار باشد، تابع assign_task_to_Cpu را صدا میزنیم. در غیر اینصورت چک میکنیم که تسک در حال پردازش تمام شده است یا خیر.برای اینکار زمان شروع پردازش را با مدت زمان اجرای تسک جمع میکنیم.اگر زمان حاضر بزرگتر مساوی این مقدار باشد، یعنی تسک تمام شده است و با استفاده از تسک وزمان شروع تسک یک instance از کلاس DoneTask می سازیم و آن را در ارایه cpu done_tasks ذخیره میکنیم.و در اخر تابع assign_task_to_cpu را صدا میزنیم تا پردازش تسک بعدی شروع شود.
سپس چک میکنیم که ایا مدت زمان انتظار هر تسک موجود در صف از مقدارtimeout بیشتر شده است یا خیر.اگر بیشتر شده باشد، آن تسک را حذف میکنیم.
در اخر تعداد موجود تسک های در صف در آن ثانیه را در ارایه cpu length_queue ذخیره میکنیم.

```python
for time in range(time_of_simulation):
    if time % time_period == 0:
        JobLoading(time)

    # UPDATING TIME_QUEUE FOR TASKS
    for task in FCFS.tasks:
        task.time_queue += 1
    for task in Round_Robin_T1.tasks:
        task.time_queue += 1
    for task in Round_Robin_T2.tasks:
        task.time_queue += 1
    for task in priority_queue.tasks:
        task.time_queue += 1

    if cpu.assigned_task is None:
        assign_task_to_CPU(time)
    else:
        cpu.time_working += 1
        # CHECK WHETHER ASSIGNED TASK TO CPU IS FINISHED
        if time >= cpu.time_start_task + cpu.assigned_task.service_time:
            cpu.done_tasks.append(DoneTask(cpu.assigned_task, cpu.time_start_task))
            cpu.assigned_task = None
            assign_task_to_CPU(time)
```

```
    # CHECKING WHETHER TIMEOUT IS HAPPENED
    for task in FCFS.tasks:
        if task.time_queue > time_out:
            FCFS.tasks.remove(task)
        else:
            break
    for task in Round_Robin_T1.tasks:
        if task.time_queue > time_out:
            Round_Robin_T1.tasks.remove(task)
        else:
            break
    for task in Round_Robin_T2.tasks:
        if task.time_queue > time_out:
            Round_Robin_T2.tasks.remove(task)
        else:
            break
    for task in priority_queue.tasks:
        if task.time_queue > time_out:
            priority_queue.tasks.remove(task)
        else:
            break
    cpu.length_queue.append(len(FCFS.tasks) + len(Round_Robin_T1.tasks) + len(Round_Robin_T2.tasks))
```

تولید خروجی

خروجی های خواسته شده پروژه را محاسبه و پرینت میکنیم.

```
print("************************************************")
print(f"Average length of queues = {sum(cpu.length_queue) / time_of_simulation} s")
print("************************************************")
print("Time spent in queues:")
time_queues = []
for done_task in cpu.done_tasks:
    time_queues.append(done_task.task.time_queue)
print(f"    ", end='')
if len(time_queues) == 0:
    print("empty")
else:
    print(*time_queues)
    print(f"    Average = {sum(time_queues) / len(time_queues)} s")
print("************************************************")
print(f"CPU efficiency = {cpu.time_working / time_of_simulation} %")
print("************************************************")
print("If we increase T1 and T2, average time spent in queues would be less.")
print("************************************************")
print(f"Dumped tasks rate = {(count_tasks - len(cpu.done_tasks)) * 100 / count_tasks} %")
```