

به نام خدا

گزارش پروژه شبیه سازی کامپیوتری

محمد مولوی - ۹۹۱۰۵۷۵۳

محمد جواد ماهرالنقش - ۹۹۱۰۵۶۹۱

استاد: دکتر بردیا صفایی

تابستان ۱۴۰۲

فهرست موضوعات

۳.....	مقدمه
۴.....	ساختار کلی
۵.....	نحوه پیاده سازی
۶.....	جزئیات کد
۱۵.....	خروجی ها و نتیجه گیری
۲۱.....	منابع و تشکرها

مقدمه

در این پروژه به پیاده‌سازی چند سیستم صف میپردازیم که به ۳ سیاست مختلف رفتار خواهند کرد. همچنین از ۲ هاست و ۱ روتر استفاده خواهیم کرد.

در شبکه، بسته‌ها می‌خواهند از مبدا به مقصد برسند، در این مسیر از روتر عبور می‌کنند.

در این پروژه می‌خواهیم رفتار روتر و تحولات بسته‌ها را بررسی کنیم و همچنین مواردی را در پایان مصورسازی (Visualize) کنیم و همچنین تمام موارد را باید شبیه‌سازی کنیم، یعنی زمان‌های میان‌ورودی (interarrival times)، زمان‌های اجرایی بسته‌ها (execution times) و همچنین فرایند تخصیص بسته‌ها به هسته‌های پردازشی باید تماماً شبیه‌سازی شوند.

برای انجام این کار از زبان پایتون استفاده کردیم و کدها همزمان در فایل پایتون و فایل جویپتر نوتبوک موجود هستند.

همچنین گزارشی از کارهای انجام‌شده در پایان نمودارسازی شده و همچنین در فایل اکسل (CSV) موارد مرتبط ذخیره میشوند.

جزئیات بیشتر را در ادامه گزارش بررسی میکنیم.

این پروژه، ذیل پروژه درس شبیه‌سازی کامپیوتری تعریف شده و در تابستان ۱۴۰۲ توسط تیم ما پیاده‌سازی شد.

ساختار پروژه

- بسته (Packet)

این موجودیت با کلاس Task پیاده‌سازی شده است. هر بسته‌ای که وارد سیستم می‌شود، مانند وظیفه‌ای است که باید به آن رسیدگی شود. هر بسته دارای ویژگی‌های مختلفی است که نیاز است تا برای بهینگی پیاده‌سازی برای آن کلاس جداگانه تعریف کنیم. اولویت، زمان ورود، و... از مواردی است که در این کلاس ذخیره می‌شوند. توابع (methods) نیز در این کلاس تعریف شده که در ادامه گزارش بررسی می‌شوند.

- ساختار کلی صف (BaseQueue)

در اینجا نیاز است تا ما یک ساختار کلی را برای صف در این سیستم طراحی کنیم. البته صف جزئیات بیشتری دارد که در کلاس‌هایی که از این کلاس ارث‌بری می‌کنند تعریف می‌شوند.

توجه کنید که برای استانداردسازی پیاده‌سازی و همچنین خوانایی بهتر، تمام سیستم‌های صف که در ادامه معرفی می‌شوند از این کلاس صف ارث‌بری می‌کنند.

- سیاست‌های صف

۱. سیاست FIFO: برای این سیاست، یک کلاس به نام FIFO تعریف شده است که از کلاس سیاست کلی صف ارث‌بری می‌کند و بسته بعدی که باید اجرا شود را به ما برمی‌گرداند و همچنین می‌توانیم زمان ورودی بسته‌های جدید را به آن اضافه کنیم.

۲. سیاست WRR: برای این سیاست نیز کلاسی به همین نام تعریف کردیم که مشابه از کلاس صف اصلی ارث‌بری می‌کند. مشابه دو تا تابع قبلی در اینجا نیز پیاده‌سازی شده است.

۳. سیاست NPP: برای این سیاست نیز دقیقاً کارهای مشابه توضیح داده شده انجام شده است.

- روتر (Router)

در این کد و شبیه‌سازی، اصل کار در واقع همین روتر است که دو Host را به یکدیگر متصل می‌کند و وظیفه مدیریت رویدادها را دارد. رویدادها در ادامه تعریف می‌شوند. بر اساس رویدادهای سیستم، اختصاص بسته‌ها و پردازش‌های مربوط به آن انجام می‌شوند. تابع اصلی که کل کارها را انجام می‌دهد نیز execute_all_tasks است که در ادامه به طور دقیق بررسی می‌شوند.

- رویداد (Event)

برای جلوگیری از تداخل در شبیه‌سازی چند راه را می‌توان پیش گرفت:

۱. زمان را با یک اپسیلون زمانی جلو ببریم، مثلاً زمان را ۰.۰۰۱ ثانیه - ۰.۰۰۱ ثانیه جلو ببریم. این روش مشخصاً بهینه نیست چراکه در بسیاری از زمان‌ها هیچ کاری قرار نیست کاری انجام شود. همچنین پیاده‌سازی کد نیز سخت می‌شود.

۲. زمان را بر اساس یک سری رویداد جلو ببریم. رویداد (Event) از ۲ نوع است:

۱.۲ حداقل یک بسته‌ای بوده که اجرایش تمام شده است و در نتیجه باید سراغ صف برویم و در صورتی که صف خالی نیست، بر اساس سیاست اولویت‌دهی‌ای که داریم به آن پردازنده خالی شده یک بسته اختصاص دهیم.

۲.۲ بسته جدیدی وارد سیستم می‌شود که نیاز است تا بررسی شود که این بسته جدید وارد شده را به کجا اختصاص دهیم.

از این ۲ سیاست گفته شده، ما مورد دوم را پیاده‌سازی کرده‌ایم و در نتیجه نیاز است که موجودیتی (کلاسی) به نام EventType تعریف کنیم که ۲ مقداری بگیرد که به معنای هر یک از این ۲ رویداد معرفی شده در ۱.۲ و ۲.۲ است.

نحوه پیاده سازی

۱. اعداد مربوط به زمان های میان ورودی تولید می شوند (از توزیع پواسون).
۲. اعداد مربوط به زمان های اجرا تولید می شوند (از توزیع نمایی).
۳. ایجاد روتر و صدا زدن تابع `execute_all_tasks` از همین کلاس.
۴. تعیین تمام زمان های ورودی بر اساس زمان های میان ورود به کمک تابع `set_all_arrivals`.
۵. تا وقتی که زمان شبیه سازی تمام نشده، کارهایی که در ادامه می آیند انجام شود و در غیر این صورت شبیه سازی را تمام کنیم (از گام ۹ ادامه دهیم).
۶. نوع رویداد بعدی و زمان رویداد بعدی را به کمک تابع `handle_and_get_next_event` بگیریم.
۷. اگر نوع رویداد از نوع تمام شدن زمان اجرایی بسته است، پردازنده خالی را به کمک تابع `get_first_free_processor` بگیریم. سپس تسک بعدی را به کمک `service_policy.get_next` بگیریم و در صورتی که تسکی موجود بود، به کمک `execute` آن را اجرا کنیم.
۸. اگر نوع رویداد از نوع آمدن بسته (تسک) جدید بود، اولین پردازنده خالی به کمک `get_first_free_processor` دریافت می شود و در صورتی که چنین پردازنده ای نبود (همه پردازنده ها مشغول بودند)، آن را وارد صف می کنیم و در غیر این صورت آن را به کمک `execute` اجرا می کنیم.
۹. در صورتی که در گام ۵ از زمان شبیه سازی فراتر رفتیم، از اینجا ادامه می دهیم و تابع `finish_all` را صدا می کنیم.
۱۰. در صورتی که تسکی شروع به اجرا کرده ولی تمام نشده (به دلیل اتمام زمان شبیه سازی)، آن را به کمک تابع `finish` تمام می کنیم.

جزئیات کد

در ادامه در هر بخش، جزئیات نحوه پیاده‌سازی توابع را بررسی خواهیم کرد.

۱. موجودیت Task (بسته)

در اینجا می‌خواهیم کلاس Task که مربوط به بسته‌ها است را بررسی کنیم.

در هنگام نمونه‌گیری (instance گیری) از این کلاس، وارد `__init__` شده و ویژگی‌های مربوط به آن مقداردهی می‌شوند.

- شناسه بسته (`task_id`) به کمک یک متغیر `static` که در کلاس تعیین شده و یک شمارنده است مشخص می‌شود.
- زمان میان‌ورودی (`interarrival time`) به عنوان پارامتر ورودی داده می‌شود.
- متغیر `arrival` نیز در ابتدا خالی گذاشته می‌شود تا بعداً که زمان ورود را تعیین کردیم آن را نیز درست کنیم.
- اولویت (`priority`) نیز یکی از متغیرهای ورودی است که مشخص می‌شود بسته از کدام یک از ۳ اولویت است.
- زمان اجرا (`execution_time`) نیز از متغیرهای ورودی است و زمانی است که اجرایی بسته طول می‌کشد.
- زمان شروع به اجرا (`start_execution_time`) در ابتدا خالی گذاشته می‌شود تا در ادامه زمان شروع اجرا را تعیین کنیم (بستگی به پردازنده‌های خالی و وظایف (بسته‌های) دیگر دارد.
- پردازنده‌ای که قرار است بر روی آن اجرا شود (`processor`) را در ابتدا خالی می‌گذاریم تا بعداً مشخص شود. دلیل خالی گذاشتن آن نیز مشابه `start_execution_time` است.
- مقدار بولین (متغیر دو‌حالتی) در ابتدا `False` گذاشته می‌شود چرا که هنوز معلوم نیست بسته زمان فراخوانی یا همان `arrival` اش رسیده باشد، در واقع باید منتظر بمانیم تا وارد سیستم شود.
- در انتها، شمارنده مربوط به تعداد بسته‌ها (`TASK_ID`) را یکی زیاد می‌کنیم که در واقع یک `Auto_Counter` است.

```
class FinishProgramException(Exception):  
    pass
```

```
class Task:  
    TASK_ID = 1  
  
    def __init__(self, inter_arrival=None, priority=None, execute_time=None):  
        self.task_id = Task.TASK_ID  
        self.inter_arrival = inter_arrival  
        self.arrival = None  
        self.priority = priority  
        self.execution_time = execute_time  
        self.start_execution_time = None  
        self.end_execution_time = None  
        self.processor = None  
        self.is_in_queue = False  
        Task.TASK_ID += 1  
  
    def __str__(self):  
        return str(self.inter_arrival, self.priority)  
  
    @staticmethod  
    def get_next_arrival_task(all_tasks): # all tasks are sorted  
        for t in all_tasks:  
            if t.start_execution_time is None and not t.is_in_queue:  
                return t  
        raise FinishProgramException()
```

- تابع `__str__` برای این است که زمانی که می‌خواهیم از یک `instance` به عنوان رشته استفاده کنیم (مثلاً هنگام پرینت)، بدانیم چه چیزی باید `return` شود.
- تابع `get_static_method` که یک تابع `static` است (مربوط به کل کلاس است و نه یک `instance` خاص)، به ما تسک بعدی‌ای که باید اجرا شود را برمی‌گرداند. توجه کنید که برای اینکه بفهمیم چه تسکی را باید برگردانیم، بر روی تمام آنها `loop` زده و اگر تسکی زمان شروع اجراش تعیین نشده (یعنی هنوز شروع به اجرا نکرده) و در صف نیز نیست، آن را بازمی‌گرداند. اگر چنین تسکی موجود نبود نیز پایان اجرای برنامه را اعلام می‌کند (موجودیت `FinishProgramException`).

```
@staticmethod
def get_next_free_processor_task(all_tasks):
    in_progress_tasks = []
    for t in all_tasks:
        if t.start_execution_time is not None and t.end_execution_time is None:
            in_progress_tasks.append(t)
    if in_progress_tasks:
        return min(in_progress_tasks, key=lambda x: x.start_execution_time + x.execution_time)

@staticmethod
def set_all_arrivals(all_tasks):
    cum_sum = 0
    for t in all_tasks:
        t.arrival = cum_sum + t.inter_arrival
        cum_sum += t.inter_arrival

def finish(self):
    self.end_execution_time = self.start_execution_time + self.execution_time
```

ادامه موجودیت تسک را بررسی می‌کنیم که در تصویر بالا میبینیم.

- تابع `get_next_free_processor_task` به ما کمترین زمان مورد نیاز تا اتمام حداقل یکی از تسک‌ها را برمیگرداند. یعنی ابتدا تمام تسک‌های در حال اجرا را پیدا میکند بدین صورت که اگر تسکی زمان شروع به اجراش مشخص است اما زمان اتمامش مشخص نیست، یعنی هنوز تمام نشده است، پس به آرایه `in_progress` اضافه می‌کنیم. سپس میان تمام تسک‌های فعال، کمترین زمان پایان را برمیگردانیم. زمان پایان نیز به کمک جمع مقدار «زمان شروع به اجرا» و «زمان اجرا» به دست می‌آید.
- تابع `set_all_arrivals`، زمان مطلق ورود هر تسک به سیستم (`absolute arrival time`) را تعیین می‌کند. این کار به کمک زمان میان‌ورود (`interarrival times`) تعیین می‌شوند. توجه کنید که اعداد تصادفی‌ای که ما در ابتدا تولید می‌کنیم، زمان میان‌ورود است، یعنی فاصله میان ۲ زمان ورود متوالی، پس باید با جمع تمجعی (`cumulative sum`)، زمان مطلق را حساب کنیم.
- تابع `finish` نیز به اجرای تسک پایان می‌دهد. برلی پایان وظیفه نیز کافی است زمان `end_execution_time` را تعیین کنیم. همین که این مقدار دیگر `None` نباشد، یعنی اجرای تسک پایان یافته است (این را در تمام بخش‌های کد در نظر گرفته‌ایم).

۲. موجودیت `BaseQueue`

```
class BaseQueue:
    def __init__(self, length_limit):
        self.length_limit = length_limit

    def get_next(self):
        raise NotImplemented

    def add_arrival_to_queue(self, task):
        raise NotImplemented
```

این موجودیت، همانطور که در بخش‌های ابتدایی گزارش بررسی شد، صف اصلی سیستم ما است که تمام ۳ سیاست صف از آن ارث‌بری می‌کنند.

- تابع `__init__`: زمانی که `instance` لی ساخته می‌شود، صرفاً ویژگی `length_limit` با عنوان تعیین می‌شود تا اگر حجم صف پر شده بود، تسک‌های جدید `drop` شده و اضافه نشوند.
- توابع `get_next` و `add_arrival_to_queue` فعلاً به صورت `NotImplemented` هستند تا در کلاس‌های فرزندشان بسته به کارکردشان پیاده‌سازی شوند. در کلاس‌های مربوط به خودشان بررسی خواهند شد.

۳. موجودیت FIFO

- تابع `__init__`: هنگام `instance` گیری وارد این تابع می‌شویم و ابتدا تابع پدر (`BaseQueue`) را مقداردهی اولیه می‌کنیم. سپس یک صف خالی ایجاد می‌کنیم.
- تابع `get_next`: وظیفه این تابع، برگرداندن تسک بعدی از میان تسک‌های موجود در صف، بر اساس سیاست `First In First Out` یا همان `First Come First Serve` است. برای این کار در صورتی که صف خالی باشد (تسکی نداشته باشیم)، `None` برمیگردانیم، در غیر اینصورت، اولین عنصر صف را برمیگردانیم و در واقع `pop` می‌کنیم.
- تابع `add_arrival_to_queue`: وظیفه این تابع، اضافه کردن تسک جدید به صف است. البته در صورتی این اتفاق می‌افتد که به حداکثر طول صف نرسیده باشیم، که در این صورت تسک `drop` شده و اصلاً به صف اضافه نمی‌شود.

```
class FIFO(BaseQueue):

    def __init__(self, length_limit):
        super().__init__(length_limit)
        self.queue = []

    def get_next(self):
        try:
            return self.queue.pop(0)
        except IndexError:
            return None

    def add_arrival_to_queue(self, task):
        if len(self.queue) < self.length_limit:
            self.queue.append(task)
```

۴. موجودیت WRR

- تابع `__init__`: مشابه با `constructor` مربوط به `FIFO` است با این تفاوت که ۳ صف ایجاد می‌کند چرا که در الگوریتم `Weighted Round Robin` نیاز به صف‌های مجزا برای هر یک از اولویت‌ها داریم.
- تابع `get_next`: وظیفه‌اش برگرداندن تسک بعدی لی است که طبق این الگوریتم اجرا می‌شود. برای این کار، از بااولویت‌ترین صف شروع کرده (صف شماره صفر که بالاترین اولویت را دارد) و در صورتی که خالی نبود، از صف با اولویت بالاتر برداشت می‌کند.
- تابع `add_arrival_to_queue`: مشابه با همین تابع در `FIFO` است با این تفاوت که به صف مربوط به اولویت خودش اضافه می‌شود.


```

class WRR(BaseQueue):

    def __init__(self, length_limit):
        super().__init__(length_limit)
        self._priority_queues = [[] for i in range(3)]

    def get_next(self):
        for priority in range(3):
            if self._priority_queues[priority]:
                return self._priority_queues[priority].pop(0)
        return None

    def add_arrival_to_queue(self, task):
        if len(self._priority_queues[task.priority]) < self.length_limit:
            self._priority_queues[task.priority].append(task)

```

۵. موجودیت NPPS

- تابع `__init__` آن که دقیقاً مشابه با همین تابع در FIFO است.
- تابع `get_next` نیز دقیقاً همان تابع در FIFO است.
- تابع `add_arrival_to_queue` بدین گونه تعریف شده است که اگر طول صف به ماکسیمم خودش نرسیده بود، تسک را به صف اضافه می کند اما صف را نیز بر اساس اولویت و سپس بر اساس زمان ورودی مرتب میکند (در کد نهایی، زمان ورود در نظر گرفته شده نه میان ورود).

```

class NPPS(BaseQueue):
    def __init__(self, length_limit):
        super().__init__(length_limit)
        self.queue = []

    def get_next(self):
        try:
            return self.queue.pop(0)
        except IndexError:
            return None

    def add_arrival_to_queue(self, task):
        if len(self.queue) < self.length_limit:
            self.queue.append(task)
            self.queue = sorted(self.queue, key=lambda x: (x.priority, x.inter_arrival))

```

۶. موجودیت EventType

- این موجودیت در مراحل قبلی گزارش بررسی شده است، اما به طور کلی تعیین میکند که نوع رویدادی که در سیستم رخ داده چیست.

```

class EventType(Enum):
    END_TASK = 1
    NEW_TASK = 2

```

۷. موجودیت Router

- این موجودیت در بخش «نحوه پیاده‌سازی» بررسی شده است اما اینجا نیز بررسی خواهد شد.
- تابع `__init__` بدین صورت کار میکند که تمام مقادیر مربوط به روتر مقداردهی میشوند. پردازنده‌ها به تعدادی که در ورودی داده شده تولید میشوند، آرایه `busy_processors` ایجاد میشود که پردازنده‌های مشغول به کار را مشخص میکند که در ابتدا خالی است، صف بر اساس سیاستی که در ورودی تعیین شده و به حداکثر طولی که در ورودی داده شده ایجاد میشود (صف سیاست دار یکی از `FIFO – WRR – NPP`) است. مدت زمان کل شبیه‌سازی که روتر باید درگیر باشد مشخص میشود و همچنین تمام تسک‌ها و زمان فعلی (که در ابتدا صفر است) مشخص میشوند.
- تابع `handle_and_get_next_event`: ابتدا تسک بعدی لی که از راه میرسد و وارد سیستم میشود را دریافت میکنیم و زمان ورود آن را نیز ذخیره میکنیم. سپس پردازنده خالی بعدی را دریافت میکنیم. در صورتی که پردازنده خالی بعدی داشته باشیم، زمانی که خالی میشود را حساب میکنیم. در غیر اینصورت این مقدار را `None` میگذاریم. حالا زمان آن است که تابع `get_next_event_time` را فراخوانی کنیم تا بر اساس نزدیکترین زمان خالی شدن پردازنده‌ها و نزدیکترین زمان آزاد شدن تسک جدید، به ما بگوید رویداد بعدی کی قرار است رخ دهد. حال در صورتی که زمان ایونت بعدی نامشخص بود (یعنی دیگر ایونتی نداشتیم) و یا زمان ایونت بعدی از مدت زمان شبیه‌سازی ما فراتر میرفت، پایان شبیه‌سازی را اعلام میکنیم. در غیر اینصورت مطمئن هستیم که قرار است رویدادی رخ دهد و تنها لازم است که نوع آن را مشخص کنیم. در صورتی که از نوع خالی شدن پردازنده باشد، پردازنده را خالی کرده و آن را از لیست پردازنده‌های مشغول خارج میکنیم. همچنین نوع رویداد بعدی و زمان وقوع رویداد بعدی را `return` میکنیم.

```
class Router:
    def __init__(self, processors_num, service_policy, length_limit, simulation_time, all_tasks):
        self.processors = [i for i in range(processors_num)]
        self.busy_processors = []
        self.service_policy = service_policy(length_limit)
        self.length_limit = length_limit # TODO
        self.simulation_time = simulation_time
        self.all_tasks = all_tasks
        self.current_time = 0

    def handle_and_get_next_event(self):
        next_arrival_task = Task.get_next_arrival_task(self.all_tasks)
        next_arrival_time = next_arrival_task.arrival if next_arrival_task else None
        next_free_processor_task = Task.get_next_free_processor_task(self.all_tasks)
        if next_free_processor_task:
            next_free_processor_time = next_free_processor_task.start_execution_time + next_free_processor_task.execution_time
        else:
            next_free_processor_time = None
        next_event_time = self.get_next_event_time(next_arrival_time, next_free_processor_time)
        if next_event_time and next_event_time > self.simulation_time or next_event_time is None:
            raise FinishProgramException()
        if next_event_time == next_arrival_time: # TODO what happens if next_arrival_time == next_free_processor_time
            return EventType.NEW_TASK.value, next_event_time
        elif next_event_time == next_free_processor_time:
            next_free_processor_task.finish()
            self.busy_processors.remove(next_free_processor_task.processor)
            return EventType.END_TASK.value, next_event_time
        else:
            raise Exception("what happened exactly?")
```

- تابع `get_next_event_time`: زمان رویداد بعدی را برمیگرداند. اگر هم تسک جدیدی دارد وارد میشود و هم پردازنده ای دارد خالی میشود، در این صورت مینیمم آنها را برمیگردانیم، در غیر اینصورت آن که `None` نیست را برمیگردانیم. اگر هیچ یک نبود نیز خالی برمیگردانیم.

```
def get_next_event_time(self, next_arrival_time, next_free_processor_time):
    if next_arrival_time is not None and next_free_processor_time is not None:
        return min(next_arrival_time, next_free_processor_time)
    elif next_arrival_time is not None:
        return next_arrival_time
    elif next_free_processor_time is not None:
        return next_free_processor_time
    else:
        return None
```

- کدهای اینجا نیز قبلتر بررسی شده اما مجدداً توضیحاتی داده میشود.
- تا زمانی که زمان شبیه سازی تمام نشده است، زمان را خرده خرده جلو میبریم. زمان را بر اساس رویکرد دومی که در بخش های ابتدایی گزارش توضیح دادیم جلو میبریم بدین شکل که بر اساس رویدادها (Event) جلو میرویم.
- تابع `handle_and_get_next_event` صدا زده میشود و رویداد بعدی را به همراه نوع آن دریافت میکند.
- در صورتی که رویداد از جنس ورود تسک جدید بود، به دنبال یک پردازنده جدید میگردیم تا آن تسک را به آن پردازنده اختصاص دهیم.
- تسک را نیز به کمک `get_next_arrival_task` دریافت میکنیم.
- در صورتی که پردازنده ای موجود بود، اجرای تسک را به کمک `execute` شروع میکنیم.
- در صورتی که پردازنده ای موجود نبود، تسک را بر اساس `policy` ای که داریم (یکی از FIFO-WRR-NPPS) به صف اضافه میکنیم (طبعاً اگر ظرفیت صف پر نشده باشد).
- در صورتی که رویداد از نوع خالی شدن پردازنده باشد، شماره پردازنده را دریافت کرده و در صورتی که `None` بود `Exception` میدهیم چرا که وقتی وارد اینجا شدیم یعنی مطمئنیم که پردازنده ای خالی بوده اما وقتی `None` میگیریم یعنی پردازنده خالی نداریم که این یعنی `Exception`.
- در صورتی که خطا نداد، تسک بعدی ای که قرار است اجرا شود را دریافت کرده و آن را از صف گرفته (بر اساس سیاستی که داریم) و تسک را شروع به اجرا میکنیم.
- زمان فعلی را نیز به زمان رویداد بعدی به روزرسانی میکنیم.
- در صورتی که به `Exception` خوردیم نیز تمام اجرا را خاتمه میدهیم.

```

def execute_all_tasks(self):
    Task.set_all_arrivals(self.all_tasks)
    try:
        while self.current_time <= self.simulation_time:
            next_event, next_event_time = self.handle_and_get_next_event()
            if next_event == EventType.NEW_TASK.value:
                free_processor = self.get_first_free_processor()
                next_task = Task.get_next_arrival_task(self.all_tasks)
                # self.service_policy.add_arrival_to_queue(next_task)
                # next_task.is_in_queue = True
                if free_processor is not None:
                    # self.execute(self.service_policy.get_next(), free_processor, next_event_time)
                    self.execute(next_task, free_processor, next_event_time)
                else:
                    next_task.is_in_queue = True
                    self.service_policy.add_arrival_to_queue(next_task)
            elif next_event == EventType.END_TASK.value:
                free_processor = self.get_first_free_processor()
                if free_processor is not None:
                    task_in_queue = self.service_policy.get_next()
                    if task_in_queue is not None:
                        self.execute(task_in_queue, free_processor, next_event_time)
                    else:
                        raise Exception('how is it possible !?')
            else:
                raise Exception(f'next_event is not in [FINISH_PROGRAM, NEW_TASK, END_TASK]')
            self.current_time = next_event_time
    except FinishProgramException:
        self.finish_all()
    print(f'Simulation ended at {self.current_time}')

```

- در پایان این موجودیت نیز دو تابع زیر را مشاهده میکنیم.
تابع `finish_all` اجرای تمام تسک های تمام نشده را تمام میکند.
تابع `execute` نیز تسک را اجرا میکند. اجرای تسک نیز بدین معنی است که زمان شروع اجرای تسک را مشخص کنیم (همانطور که قبلتر توضیح دادیم، این بدین معنی است که اجرای این تسک شروع شده است) و همچنین پردازنده متناظر با آن را مشخص کنیم.
همچنین پردازنده ای که درگیر اجرای آن شده را به لیست پردازنده های درگیر اضافه کنیم.

```

def finish_all(self):
    for t in self.all_tasks:
        if t.start_execution_time is not None and t.end_execution_time is None:
            t.finish()

def execute(self, t, processor, next_event_time):
    t.start_execution_time = next_event_time
    t.processor = processor
    self.busy_processors.append(processor)

```

۸. اجرای شبیه سازی

- در قطعه کد زیر نیز، روند اجرا را مشاهده میکنیم که X و Y و Z همانطور که کامنت شده اند، پارامترهای توزیع پواسون، توزیع نمایی و همچنین مدت زمان شبیه سازی هستند.
- تعداد پردازنده ها، سیاست اولویت دهی به تسک ها، و حداکثر طول صف نیز تعیین میشود.
- سپس زمان های میان ورود و اجرا از توزیع های مناسب خواسته شده تولید شده و packets تولید میشود.
- توجه کنید که تعداد بسته ها را بیش از میزان مورد نیاز در نظر گرفتیم تا هیچ گاه این اتفاق نیفتد که شبیه سازی زودتر از موعد تمام شود.
- البته برای اینکه اضافات این بسته ها دور ریخته شود، یک loop زده و تا زمانی که از زمان شبیه سازی فراتر نمیرویم این بسته ها را اضافه میکنیم.
- در نهایت Router را ایجاد کرده و شبیه سازی را شروع میکنیم.

```
X = 3 # parameter of the poisson distribution (in packet generation)
Y = 6 # parameter of the exponential distribution (in router - for service time generation)
T = 500 # Total simulation time

PROCESSORS_NUM = 1 # It can vary
SERVICE_POLICY = [FIFO, WRR, NPPS][2] # It can vary
LENGTH_LIMIT = 20 # It can vary

generator = np.random.default_rng()
packet_arrivals = generator.poisson(lam=X, size=T * X * 2)
priorities = np.random.choice([0, 1, 2], p=[0.2, 0.3, 0.5], size=T * X * 2)

generator = np.random.default_rng()
packet_times = generator.exponential(Y, size=T * X * 2)

arrivals = 0
packets = []
counter = 0
while arrivals + packet_arrivals[counter] <= T:
    packets.append(
        Task(inter_arrival=packet_arrivals[counter], priority=priorities[counter], execute_time=packet_times[counter]))
    arrivals += packet_arrivals[counter]
    counter += 1

r = Router(processors_num=PROCESSORS_NUM, service_policy=SERVICE_POLICY, length_limit=LENGTH_LIMIT, simulation_time=T,
            all_tasks=packets)
r.execute_all_tasks()
```

۹. در نهایت مطابق با قطعه کد زیر، اطلاعات شبیه سازی در فایل CSV ذخیره میشوند.

- به ازای تمام تسک ها، شناسه تسک، زمان میان ورود، زمان ورود، اولویت، زمان اجرا، زمان پایان اجرا، شماره پردازنده ای که بر روی آن اجرا شده، و اینکه آیا اصلا به روتر ارسال شده یا نه، و اینکه آیا تسکی بوده که به دلیل پر بودن صف drop شده، همگی در فایل اکسل ذخیره میشوند.

```
import csv

# Define a list of tasks
tasks = packets

# Open a CSV file for writing
with open('tasks.csv', 'w', newline='') as csvfile:
    # Create a CSV writer object
    writer = csv.writer(csvfile)

    # Write the header row
    writer.writerow(
        ['task_id', 'inter_arrival', 'arrival', 'priority', 'execution_time', 'start_execution_time',
         'end_execution_time', 'processor', 'gone_to_router', 'dropped'])

    # Write the data rows
    for task in tasks:
        writer.writerow(
            [task.task_id, task.inter_arrival, task.arrival, task.priority, task.execution_time,
             task.start_execution_time, task.end_execution_time, task.processor, task.gone_to_router, task.dropped])
```

خروجی‌ها و نتیجه‌گیری

در پایان، خروجی‌های خواسته شده رسم میشوند.

۱. متوسط طول صف

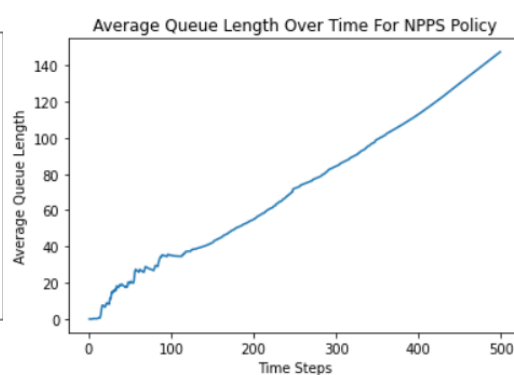
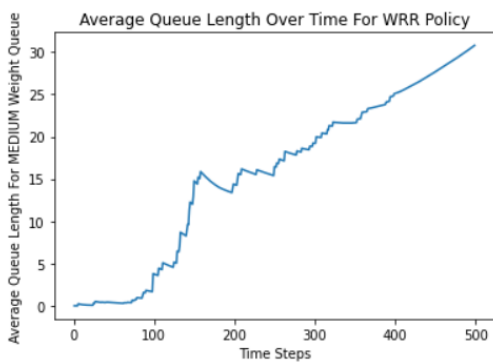
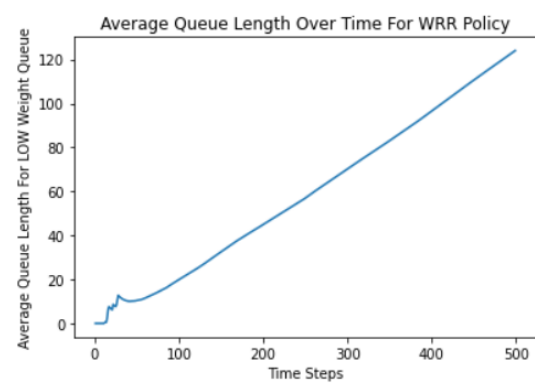
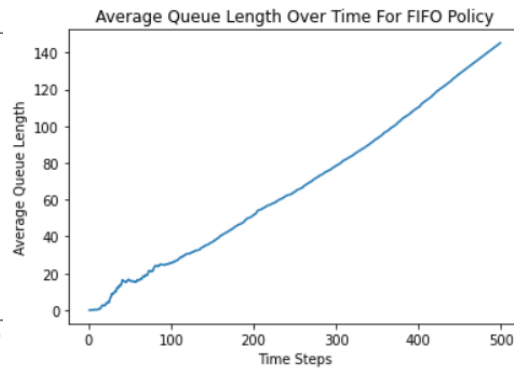
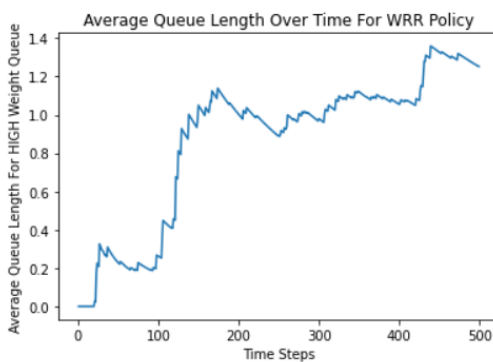
۱.۱. کد

Mean Length of the Queue

```
def plot_mean_length_of_queue(tasks, service_policy):
    if service_policy == 'WRR':
        for priority_key, priority_label in priority_labels.items():
            queue_times = []
            time_steps = []
            for i in range(1, T):
                queue_times.append(Task.get_cumulative_queue_time(tasks, i, priority_key) / i)
                time_steps.append(i)
            plt.plot(time_steps, queue_times)
            plt.xlabel('Time Steps')
            plt.ylabel(f'Average Queue Length For {priority_label} Weight Queue')
            plt.title(f'Average Queue Length Over Time For {service_policy} Policy')
            plt.show()
    else:
        queue_times = []
        time_steps = []
        for i in range(1, T):
            queue_times.append(Task.get_cumulative_queue_time(tasks, i, None) / i)
            time_steps.append(i)
        plt.plot(time_steps, queue_times)
        plt.xlabel('Time Steps')
        plt.ylabel('Average Queue Length')
        plt.title(f'Average Queue Length Over Time For {service_policy} Policy')
        plt.show()

plot_mean_length_of_queue(fifo_packets, 'FIFO')
plot_mean_length_of_queue(npps_packets, 'NPPS')
plot_mean_length_of_queue(wrr_packets, 'WRR')
```

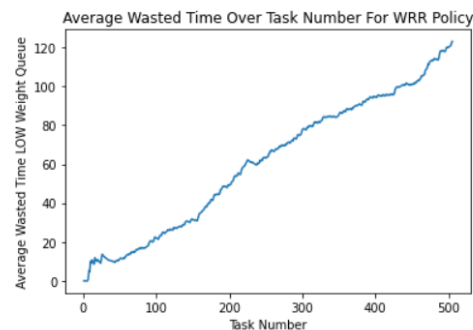
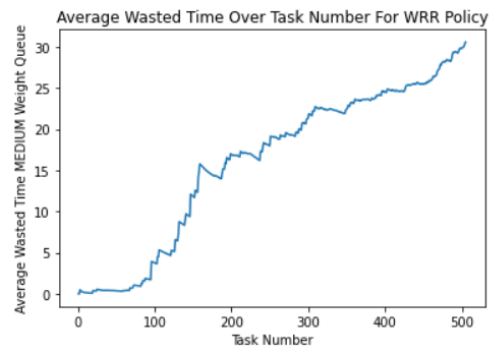
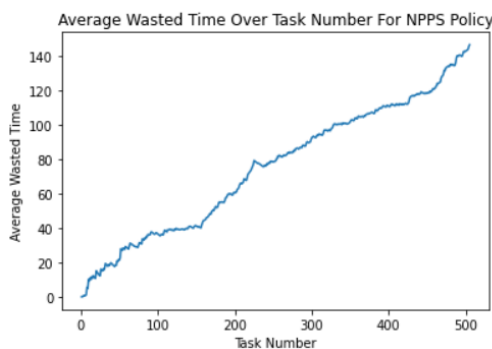
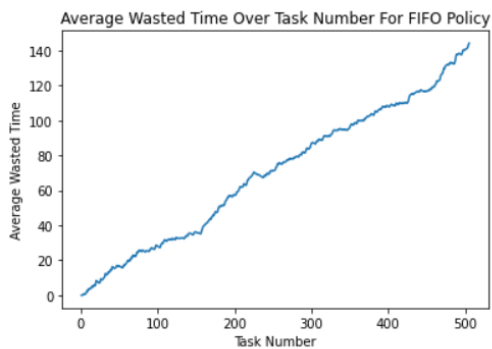
۲.۱. خروجی



Mean time wasted in all Queues

```
def plot_mean_time_wasted_in_queue(tasks, service_policy):
    if service_policy == 'WRR':
        for priority_key, priority_label in priority_labels.items():
            queue_times = []
            task_number = []
            for i in range(1, len(tasks) + 1):
                queue_times.append(Task.get_cumulative_queue_time(tasks, tasks[i - 1].arrival, priority_key) / i)
                task_number.append(i)
            plt.plot(task_number, queue_times)
            plt.xlabel('Task Number')
            plt.ylabel(f'Average Wasted Time {priority_label} Weight Queue')
            plt.title(f'Average Wasted Time Over Task Number For {service_policy} Policy')
            plt.show()
    else:
        queue_times = []
        task_number = []
        for i in range(1, len(tasks) + 1):
            queue_times.append(Task.get_cumulative_queue_time(tasks, tasks[i - 1].arrival, None) / i)
            task_number.append(i)
        plt.plot(task_number, queue_times)
        plt.xlabel('Task Number')
        plt.ylabel('Average Wasted Time')
        plt.title(f'Average Wasted Time Over Task Number For {service_policy} Policy')
        plt.show()

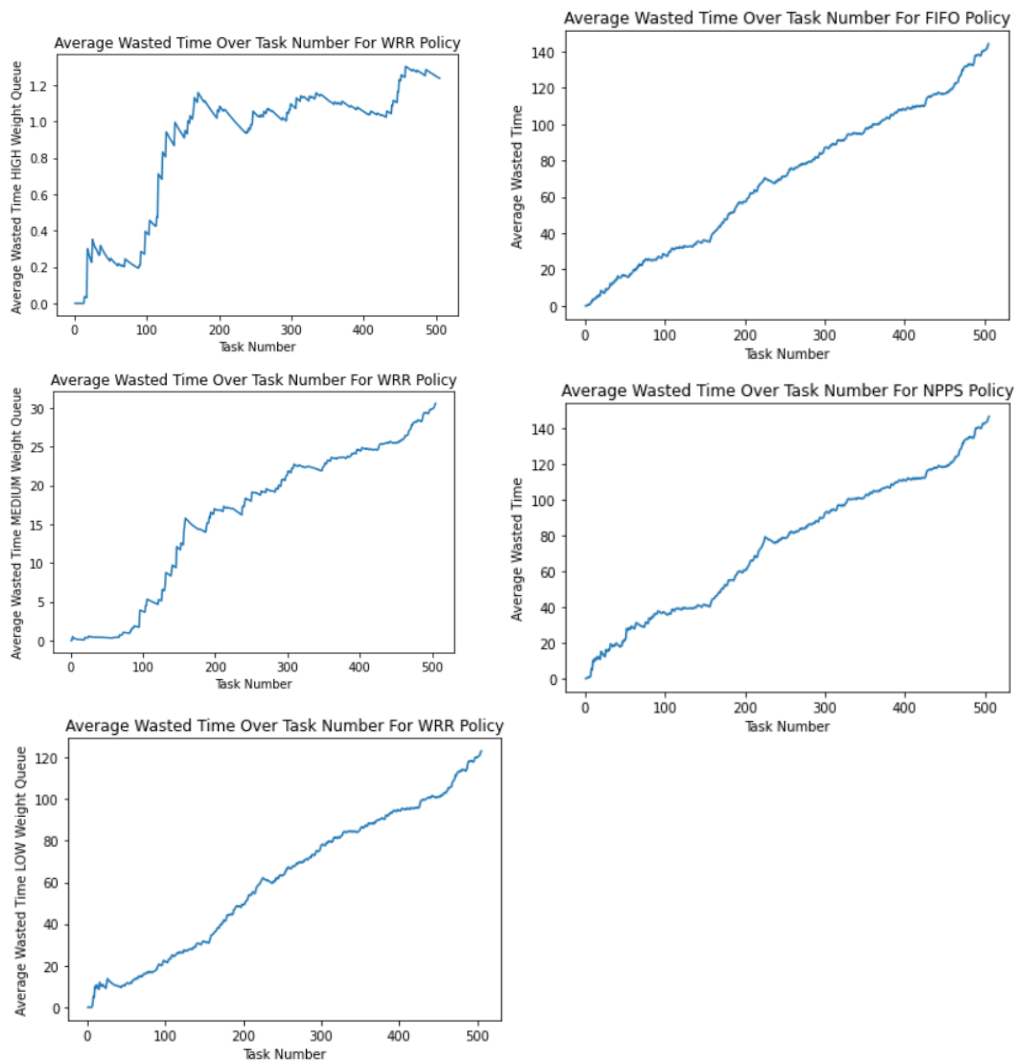
plot_mean_time_wasted_in_queue(fifo_packets, 'FIFO')
plot_mean_time_wasted_in_queue(npps_packets, 'NPPS')
plot_mean_time_wasted_in_queue(wrr_packets, 'WRR')
```



Mean time wasted in each Queue

```
def plot_mean_time_wasted_in_queue(tasks, service_policy):
    if service_policy == 'WRR':
        for priority_key, priority_label in priority_labels.items():
            queue_times = []
            task_number = []
            for i in range(1, len(tasks) + 1):
                queue_times.append(Task.get_cumulative_queue_time(tasks, tasks[i - 1].arrival, priority_key) / i)
                task_number.append(i)
            plt.plot(task_number, queue_times)
            plt.xlabel('Task Number')
            plt.ylabel(f'Average Wasted Time {priority_label} Weight Queue')
            plt.title(f'Average Wasted Time Over Task Number For {service_policy} Policy')
            plt.show()
    else:
        queue_times = []
        task_number = []
        for i in range(1, len(tasks) + 1):
            queue_times.append(Task.get_cumulative_queue_time(tasks, tasks[i - 1].arrival, None) / i)
            task_number.append(i)
        plt.plot(task_number, queue_times)
        plt.xlabel('Task Number')
        plt.ylabel('Average Wasted Time')
        plt.title(f'Average Wasted Time Over Task Number For {service_policy} Policy')
        plt.show()

plot_mean_time_wasted_in_queue(fifo_packets, 'FIFO')
plot_mean_time_wasted_in_queue(npps_packets, 'NPPS')
plot_mean_time_wasted_in_queue(wrr_packets, 'WRR')
```

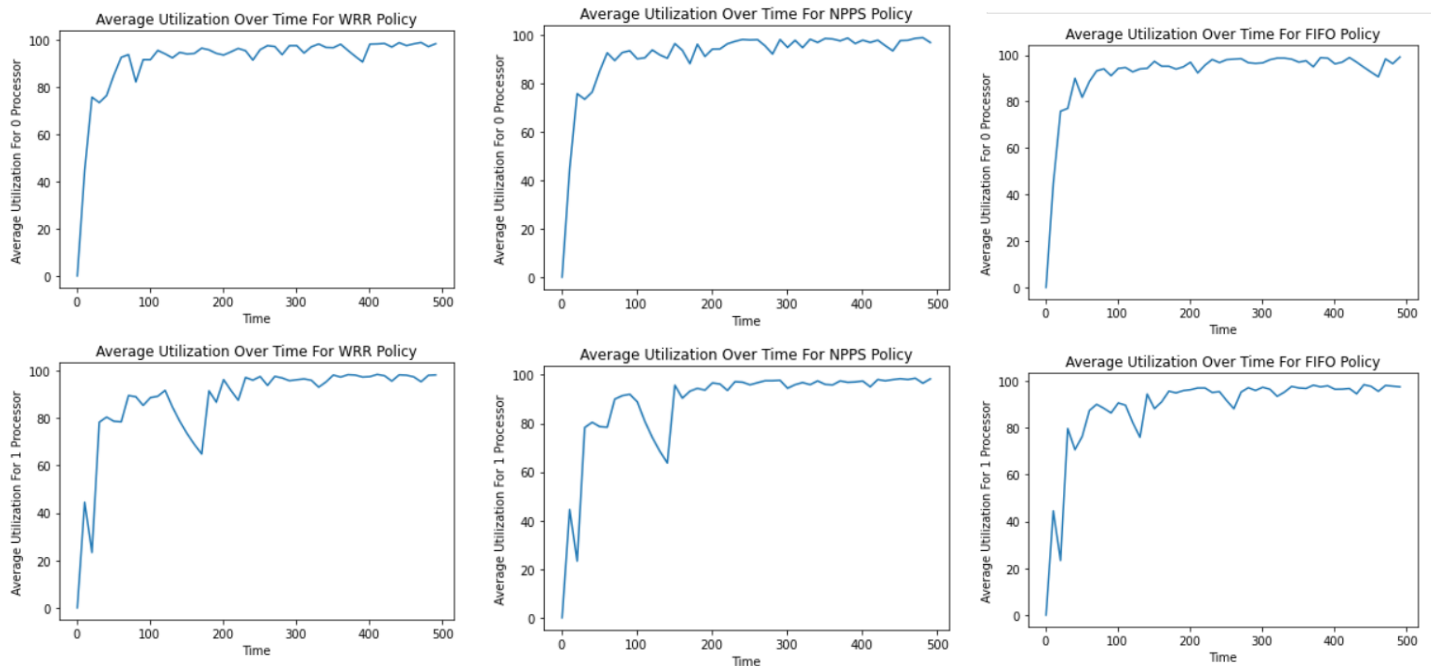


Mean Utilization of each processor

```
def plot_mean_utilization_for_processor(tasks, service_policy):
    for processor in range(PROCESSORS_NUM):
        utilization_percentages = []
        task_number = []
        for i in range(1, T, 10):
            utilization_percentages.append(Task.get_cumulative_execution_time(tasks, i, processor) / i * 100)
            task_number.append(i)
        plt.plot(task_number, utilization_percentages)
        plt.xlabel('Time')
        plt.ylabel(f'Average Utilization For {processor} Processor')
        plt.title(f'Average Utilization Over Time For {service_policy} Policy')
        plt.show()

plot_mean_utilization_for_processor(fifo_packets, 'FIFO')
plot_mean_utilization_for_processor(npps_packets, 'NPPS')
plot_mean_utilization_for_processor(wrr_packets, 'WRR')
```

۴.۲. خروجی



Number of dropped packages

```
print(Task.get_number_of_dropped_tasks(fifo_packets), 'are dropped from FIFO Policy')
print(Task.get_number_of_dropped_tasks(wrr_packets), 'are dropped from WRR Policy')
print(Task.get_number_of_dropped_tasks(npps_packets), 'are dropped from NPPS Policy')
```

301 are dropped from FIFO Policy
291 are dropped from WRR Policy
304 are dropped from NPPS Policy

۶. نمودارهای CDF مربوط به تسک های با اولویت بالا

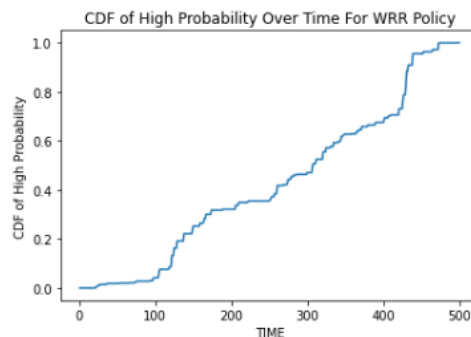
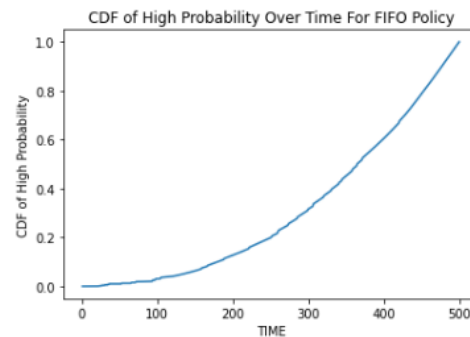
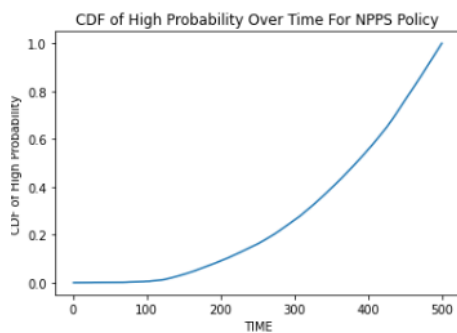
CDF Plots of high-priority Packages

۱.۶. کد

```
def plot_cdf_time_wasted_in_queue(tasks, service_policy):
    queue_cdf = []
    task_number = []
    for i in range(1, T + 1):
        queue_cdf.append(Task.get_cumulative_queue_time(tasks, i, 0) / (T * len(tasks)))
        task_number.append(i)
    normalizer = 1 / queue_cdf[-1]
    for i in range(len(queue_cdf)):
        queue_cdf[i] *= normalizer
    plt.plot(task_number, queue_cdf)
    plt.xlabel('TIME')
    plt.ylabel('CDF of High Probability')
    plt.title(f'CDF of High Probability Over Time For {service_policy} Policy')
    plt.show()
```

```
plot_cdf_time_wasted_in_queue(fifo_packets, 'FIFO')
plot_cdf_time_wasted_in_queue(wrr_packets, 'WRR')
plot_cdf_time_wasted_in_queue(npps_packets, 'NPPS')
```

۲.۶. خروجی



۷. چگونه میتوانیم utilization سیستم را بالا ببریم؟

* تقسیم سنگینی (Load Balancing): تقسیم کردن مناسب تسک ها میان پروسورها

* پردازش موازی: از مولتی ترد و چندپردازشی استفاده کنیم تا از ریسورس های موجود حداکثر استفاده را بکنیم.

* بهینه سازی منابع: اختصاص منابع را بهینه سازی کنیم. این کار به کمک تحلیل و بررسی عملکرد سیستم و مشخص کردن ریسورس هایی که کم یا زیاد از حد استفاده شده اند به دست می آید، بدین صورت میتوانیم توازن بهتری در تقسیم تسک ها نیز داشته باشیم که به همان Load Balancing نیز مرتبط میشود.

* الگوریتم های زمانبندی: از الگوریتم های مختلف زمانبندی یا همان Task Scheduling های مختلف استفاده کنیم تا اولویت بندی مناسب کرده و بتوانیم زمان های بیکاری را کمینه و استفاده از منابع را بیشینه کنیم. این کار به کمک در نظر گرفتن مواردی همچون وابستگی ها، اولویت ها و پیش نیازها قابل دستیابی است.

۸. با توجه به داده های به دست آمده، با در نظر گرفتن دو معیار تعداد بسته هایی که drop میشوند و تعداد دسته هایی که به دست host میرسند، کدام یک از سیاست های نوبت دهی در روتر منطقی تر و بهتر است که استفاده کنیم؟ توضیح دهید.

سیاست WRR بهتر است چرا که بسته های کمتری را drop میکند و سایر نتایج و خروجی های مقایسه ای نیز در نمودارهای بالا نمایش داده شده اند.

منابع و تشکرها

برای انجام این پروژه از مطالب درس شبیه سازی که در کلاس گفته شد استفاده کردیم .

همچنین جست و جویهای مختلفی نیز انجام دادیم و در صورت نیاز از گروه های دیگر نیز مشورت و همفکری گرفتیم .

با تشکر از استاد و تیم تدریس که ما را در انجام این پروژه یاری کردند .