به نام خدا

گزارش پروژه شبیه‌سازی کامپیوتری

محمد مولوی - ۹۹۱۰۵۷۵۳

محمدجواد ماهرالنقش – ۹۹۱۰۵۶۹۱

استاد: دکتر بردیا صفایی

تابستان ۱۴۰۲

فهرست موضوعات

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 می کنند.

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 اجرای بسته‌ها (execution times) و همچنین فرایند تخصیص بسته‌ها به هسته‌های پردازشی باید تماما شبیه‌سازی شوند.

برای انجام این کار از زبان پایتون استفاده کردیم و کدها همزمان در فایل پایتون و فایل جوپیتر 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 ۱۴۰۱ 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 مشابها از کلاس صف اصلی ارث‌بری می‌کند. مشابها دو تا تابع قبلی در اینجا نیز پیاده‌سازی شده است.

۳. سیاست NPP: برای این سیاست نیز دقیقا کارهای مشابه توضیح‌داده‌شده 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 جلوبردن زمان در شبیه‌سازی چند راه را می‌توان پیش گرفت:

۱. زمان را با یک اپسیلون زمانی جلو ببریم، مثلا زمان را ۰.۰۰۰۱ ثانیه – ۰.۰۰۰۱ ثانیه جلو ببریم. این روش مشخصا 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 خالی‌شده یک بسته اختصاص دهیم.

۲.۲. بسته جدیدی وارد سیستم می‌شود که نیاز است تا بررسی شود که این بسته جدید واردشده 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 پیاده‌سازی

۱. اعداد مربوط به زمان‌های میان‌ورودی تولید می‌شوند (از توزیع پوآسون).
۲. اعداد مربوط به زمان‌های اجرا تولید می‌شوند (از توزیع نمایی).
۳. ایجاد روتر و صدا زدن تابع execute_all_tasks از همین کلاس.
۴. تعیین تمام زمان‌های ورودی بر اساس زمان‌های میان‌ورود به کمک تابع set_all_arrivals.
۵. تا وقتی که زمان شبیه‌سازی تمام نشده، کارهایی که در ادامه می‌آیند انجام شود و در غیراینصورت شبیه‌سازی 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 تمام‌شدن زمان اجرای بسته است، پردازندهٔ 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 می‌شود و در صورتی که چنین پردازنده‌ای نبود (همه پردازنده‌ها مشغول بودند)، آن را وارد صف می کنیم و در غیراینصورت آن را به کمک execute اجرا می کنیم.
۹. در صورتی که در گام ۵ از زمان شبیه‌سازی فراتر رفتیم، از اینجا ادامه می‌دهیم و تابع finish_all را صدا می کنیم.
۱۰. در صورتی که تسکی شروع به اجرا کرده ولی تمام نشده (به دلیل اتمام زمان شبیه‌سازی)، 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 ویژگی های 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 بعدا 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 اجرای بسته طول می کشد.
- 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 می گذاریم تا بعدا مشخص شود. دلیل خالی گذاشتن آن نیز مشابه start_execution_time است.
- مقدار بولین (متغیر دو حالتی) 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 می کنیم که در واقع یک Auto_Counter است.

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

```python
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 (مثلا 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 برمی گرداند. توجه کنید که برای اینکه بفهمیم چه تسکی را باید برگردانیم، بر روی تمام آنها loop زده و اگر تسکی زمان شروع اجرایش تعیین نشده (یعنی هنوز شروع به اجرا نکرده) و در صف نیز نیست، آن را بازمی گرداند. اگر چنین تسکی موجود نبود نیز پایان اجرای برنامه را اعلام می کند (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 تسک ها را برمیگرداند. یعنی ابتدا تمام تسک های در حال اجرا را پیدا میکند بدین صورت که اگر تسکی زمان شروع به اجرایش 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 می کند. این کار به کمک زمان میان‌ورود (interarrival times) تعیین می شوند. توجه کنید که اعداد تصادفی‌ای که ما در ابتدا تولید می کنیم، زمان میان‌ورود است، یعنی فاصله میان ۲ زمان ورود متوالی، پس باید با جمع تجمعی (cumulative sum)، زمان مطلق را حساب کنیم.
- تابع finish نیز به اجرای تسک پایان می‌دهد. برای 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 می کنند.

- تابع __init__: زمانی که instance ای ساخته می‌شود، صرفا ویژگی ای با عنوان length_limit تعیین می‌شود تا اگر حجم صف پر شده بود، تسک‌های جدید drop شده و اضافه نشوند.
- توابع get_next و add_arrival_to_queue فعلا 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 تسک های 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 غیراینصورت، 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 اصلا به صف اضافه نمی‌شود.

```python
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 می کند چراکه در الگوریتم Weighted Round Robin نیاز به صف‌های مجزا برای هر یک از اولویت‌ها داریم.
- تابع get_next: وظیفه‌اش برگرداندن تسک بعدی‌ای 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 می کند.
- تابع add_arrival_to_queue: مشابه با همین تابع در FIFO است با این تفاوت که به صف مربوط به اولویت خودش اضافه می‌شود.

```python
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 دقیقا مشابه با همین تابع در FIFO است.
- تابع get_next نیز دقیقا 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.

```python
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.

```python
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. پردازنده ها به تعدادی که در ورودی داده شده تولید میشوند، آرایه busy_processors ایجاد میشود که پردازنده های 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 شبیه سازی که روتر باید درگیر باشد مشخص میشود و همچنی تمام تسک ها و زمان فعلی (که در ابتدا صفر است) مشخص میشوند.
- تابع handle_and_get_next_event: ابتدا تسک بعدی ای 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 پردازنده ها و نزدیکترین زمان آزادشدن 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 شبیه سازی ما فراتر میرفت، پایان شبیه سازی 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 پردازنده های مشغول خارج میکنیم. همچنین نوع رویداد بعدی و زمان وقوع رویداد بعدی را return میکنیم.

```python
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
```

```
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}')
```

ویس های محمد :

به طور کلی ۳ ساختار صف در نظر گرفتیم و هر ۳ آنها از یک کلاس ارث بری میکنند که باعث میشودد تمیسز باشد.

یک کلاس تسک یا وظیفه تعریف کردیم که تعداد ویژگی دارد برای اینکه مشخص باشد هر تسک چه ویژگی ها و مشخصاتی دارد.

یکی از مزایایی که تعریف آن به عنوان کلاس دارد...

کد اصلی در بخش Router است.

زمان را بدین صورت جلو میبریم که زمان را به اندازه مینیمم زمان خالی شدن اولین پردازنده و یا رسیدن تسکی جدید، زمان را جلو میبرد.

تا زمانی به جلو میرویم که از زمان شبیه سازی جلو نزنیم.

هر تسکی که می آید یا به پروسسور اختصاص میدهد و یا داخل صف می اندازیم.

همچنین اگر به حداثکر طول صف رسیدیم، از صف بیرون می انداختیم.

نکات

- وقتی تسک ها تمام شد، همه تسک های باقیمانده را فریز (؟) میکند.
- در پایان یک فایل csv است که در آنجا اطلاعات ذخیره میشوند.