باسمه تعالی

گزارش پروژه سیستم‌عامل

Scheduling Algorithms

وحید قرقانی ۹۶۳۲۴۶۴
سید محمدرضا موسوی

پاییز ۱۳۹۸

این پروژه در ۴ فایل با نام‌های زیر تعریف شده است:

۱. enums.py : شامل دو Enum به نام‌های Algorithm و State می‌باشد.

۲. process.py : که شامل یک Class به نام Process می‌باشد که ویژگی‌های هر پروسه و Method هایی که برای هر پروسه لازم است در این کلاس موجود است.

۳. scheduler.py : شامل Class به نام Scheduler است که فایل اصلی پروژه این قسمت می‌باشد که الگوریتم‌ها و بقیه مشخصات یک سیستم زمانبند در آن قرار دارد که در ادامه به توضیح آن خواهیم پرداخت.

۴. main.py : برای اجرای برنامه و ورودی و خروجی گرفتن از این فایل استفاده می‌کنیم.

حال به توضیح مفصل هر کدام از فایل‌ها در زیر می‌پردازیم:

enums.py :

شامل دو enum است که State وضعیت هر پروسه در هر لحظه را مشخص می‌کند که یکی از ویژگی‌های هر پروسه می‌باشد که اکنون در چه وضعیتی است.
و Algorithm نیز نشان می‌دهد زمانبندی که در حال اجرای پروسه‌ها می‌باشد اکنون از چه الگوریتمی برای زمانبندی کردن از آن استفاده می‌کند.

که بدین شرح است:

```
class State(Enum):
    NOT_ARRIVED = 0
    READY = 1
    CPU = 2
    IO = 3
    IO_TERMINATED = 4
    TERMINATED = 5


class Algorithm(Enum):
    FCFS = 0
    RR = 1
    SPN = 2
    SRT = 3
```

process.py :

این فایل که هر پروسه را بطور کامل مشخص می‌کند دارای Method هایی نیز است که به ترتیب در زیر آن‌ها را توضیح خواهیم داد:

هر پروسه ویژگی‌هایی که دارد از جمله:
۱. شماره ۲. زمان رسیدن ۳. زمان پردازش ۴. زمان I/O و ۵.وضعیت پروسه ۶. زمان تمام شدن ۷. زمان پاسخگویی ۸. زمان انتظار و...

```
def __init__(self, process_id, arrival_time, burst_time, io_time, state=State.NOT_ARRIVED):
    self.__process_id = process_id
    self.__arrival_time = arrival_time
    self.__next_arrival_time = arrival_time
    self.__burst_time = burst_time
    self.__io_time = io_time
    self.__stack = self.initialize()
    self.__state = state
    self.__response_time = -1
    self.__turn_around_time = -1
    self.__waiting_time = 0
    self.__terminate_time = 0
```

حال به توضیح متودهای این کلاس می‌پردازیم:
۱.

```
def initialize(self):
    stack = []

    if len(self.__burst_time) == len(self.__io_time):
        for i in range(len(self.__burst_time) - 1, -1, -1):
            stack.append(self.__io_time[i])
            stack.append(self.__burst_time[i])
    else:
        for i in range(len(self.__io_time) - 1, -1, -1):
            stack.append(self.__burst_time[i + 1])
            stack.append(self.__io_time[i])
        stack.append(self.__burst_time[0])

    return stack
```

از این متود برای گذاشتن زمان‌های IO و CPU بین یکدیگر استفاده می‌شود بدین صورت که زمان‌ها را یک در میان پست سر هم می‌گذارد که میان IO و CPU جابه‌جا شود.

۲.

```
def check_terminate(self):
    if not self.__stack:
        self.__state = State.TERMINATED
```

برای اینکه چک کنیم که آیا پروسه به پایان رسیده است یا نه از این متود استفاده می‌کنیم بدین صورت که stack مربوط به آن پروسه را چک می‌کند.

۳.

```
def increment_waiting_time(self):
    self.__waiting_time += 1
```

با صدا زدن این متود باعث می‌شود که زمان انتظار مربوط به هر پروسه یک واحد زیاد شود.

۴.

```
@next_arrival_time.setter
def next_arrival_time(self, time):
    if self.__state == State.IO:
        self.__stack.pop()
    self.__next_arrival_time = time
    if not self.__stack:
        self.__state = State.IO_TERMINATED
```

این متود زمانی استفاده می‌شود که پروسه از cpu خارج می‌شود و برای اینکه چه زمانی به صف ready اضاف شود زمان آن تنظیم می‌شود.

۵.

```
@property
def next_arrival_time(self):
    return self.__next_arrival_time
```

از متود برای گرفتن زمان انتظار هر پروسه استفاده می‌شود چون ویژگی‌های هر پروسه private می‌باشد برای آن تعریف شده است.

۶.

```
@property
def io_time(self):
    return self.__stack[-1]
```

این تابع getter برای اولین io روی stack می‌باشد.

۷.

```
@property
def burst_time(self):
    if self.__stack:
        return self.__stack[-1]
    else:
        return 0
```

این متود نیز شبیه متود قبل برای بدست آوردن اولین burst time می‌باشد که اگر stack خالی باشد ۰ برمی‌گرداند.

۸.

```
def minus_burst_time(self):
    self.check_terminate()
    if self.__state != State.TERMINATED:
        self.__stack[-1] -= 1
        if self.__stack[-1] == 0:
            self.__stack.pop()
            if self.__stack:
                self.__state = State.IO
        self.check_terminate()
```

این متود هر بار که تایمر یک میلی ثانیه زیاد می‌شود برای پروسه در حال اجرا صدا زده می‌شود و باعث می‌شود که از burst time آن یک میلی ثانیه کم شود و ملاحظاتی را نیز رعایت می‌کند از جمله هر بار چک می‌کند که پروسه تمام نشده باشد و اگر صفر شد آن را از روی stack مربوط به پروسه بردارد.

۹.

```
@property
def arrival_time(self):
    return self.__arrival_time
```

این متود getter برای زمان رسیدن می‌باشد که در جاهایی از جمله محاسبه response time به کار می‌آید.

۱۰.

```
@property
def state(self):
    return self.__state

@state.setter
def state(self, new_state):
    self.__state = new_state
```

این دو متود برای set کردن و get کردن وضعیت هر پروسه می‌باشد.

۱۱.

```
@property
def response_time(self):
    return self.__response_time

@response_time.setter
def response_time(self, r_time):
    self.__response_time = r_time
```

این دو متود برای set کردن و get کردن response time می‌باشد.

۱۲.

```
@property
def waiting_time(self):
    return self.__waiting_time

@waiting_time.setter
def waiting_time(self, w_time):
    self.__waiting_time = w_time
```

این دو متود برای set کردن و get کردن waiting time می‌باشد.

۱۳.

```
@property
def turn_around_time(self):
    return self.__turn_around_time

@turn_around_time.setter
def turn_around_time(self, t_time):
    self.__turn_around_time = t_time
```

این دو متود برای set کردن و get کردن turn around time می‌باشد.

۱۴.

```
@property
def terminate_time(self):
    return self.__terminate_time

@terminate_time.setter
def terminate_time(self, t_time):
    self.__terminate_time = t_time
```

این دو متود برای set کردن و get کردن terminate time می‌باشد.(زمانی که پروسه به پایان رسیده است)

scheduler.py :

این فایل شامل زمانبند است که به عبارتی مهم‌ترین فایل می‌باشد که دارای الگوریتم‌ها و صف ready و مابقی ویژگی‌های یک زمانبند می‌باشد که به صورت کامل به توضیح هر کدام از آن‌ها می‌پردازیم.

این زمانبند دارای ویژگی‌های زیر است:
۱. الگوریتم در حال اجرا ۲. همه پروسه‌ها ۳. پروسه‌های terminate شده ۴. تایمر idle.۵ یا زمانی که cpu در حال اجرای پروسه‌ای نیست ۶. Ready queue که صف پروسه‌های در وضعیت ready می‌باشد ۷.پروسه‌ای که اکنون در حال اجرا در cpu می‌باشد.

```
def __init__(self):
    self.__state = None
    self.__algorithm = None
    self.__processes = []
    self.__running_process = None
    self.__ready_queue = []
    self.__done_list = []
    self.__timer = 0
    self.__idle = 0
```

حال به ترتیب به توضیح متودها و الگوریتم‌ها می‌پردازیم:
متودها:
۱.

```
def detail(self):
    with open("/Users/VahidGh/Documents/Operating System/CPU-Scheduler/{algo_name}.txt".format(algo_name=self.__algorithm), 'w') as out_file:
        out_file.write("Average response-time = " + str(self.average_response_time()) + "\n" +
                       "Average response-time = " + str(self.average_response_time()) + "\n" +
                       "Average waiting-time = " + str(self.average_waiting_time()) + "\n" +
                       "CPU utilization = " + str(self.cpu_utilization()) + "\n" +
                       "CPU time = " + str(self.__timer) + "\n" +
                       "Idle time = " + str(self.__idle) + "\n" +
                       "-------------------------------------------------" + "\n" +
                       "#####   Processes List #####" + "\n\n"
                       )
        for process in self.__processes:
            out_file.write("Process id: " + str(process.process_id) + "\n" +
                           "Turn around time: " + str(process.turn_around_time) + "\n" +
                           "Waiting time: " + str(process.waiting_time) + "\n" +
                           "Start time: " + str(process.response_time + process.arrival_time) + "\n" +
                           "Terminate time: " + str(process.terminate_time) + "\n" +
                           "-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+" + "\n")
```

این متود جزییاتی که زمانبند بدست آورده و محاسبه کرده و همه جزییات را در یک فایل می‌نویسد که جزییات در تصویر قابل مشاهده می‌باشد به عنوان مثال جزییات هر پروسه را به صورتی که مشخص شده می‌نویسد.

.۲

```
def average_response_time(self):
    return sum([proc.response_time for proc in self.__processes]) / len(self.__processes)

def average_waiting_time(self):
    return sum([proc.waiting_time for proc in self.__processes]) / len(self.__processes)

def average_turn_around_time(self):
    return sum([proc.turn_around_time for proc in self.__processes]) / len(self.__processes)

def cpu_utilization(self):
    return ((self.__timer - self.__idle) / self.__timer) * 100

def throughput(self):
    return len(self.__processes) / self.__timer
```

این ۵ متود نیز برای محاسبه خروجی اصلی زمانبند استفاده می‌شود که میانگین زمان اجرا و زمان انتظار و ... را دارا می‌باشد.

.۳

```
@property
def processes(self):
    return self.__processes

@processes.setter
def processes(self, procs):
    self.__processes = procs
```

این متودها برای set کردن و get کردن پروسه‌های زمانبند به کار می‌رود.

.۴

```
def increment_waiting_time(self):
    for proc in self.__processes:
        if proc.state == State.READY:
            proc.waiting_time += 1
```

این متود به اینصورت عمل می‌کند که در هر میلی‌ثانیه که تایمر زیاد می‌شود به پروسه‌هایی که در صف Ready قرار دارند یک میلی‌ثانیه wait اضاف می‌کند.

۵.

```
def update_ready_queue(self):
    for proc in self.__processes:
        if proc.next_arrival_time == self.__timer and proc.state != State.IO_TERMINATED and proc not in self.__ready_queue:
            if proc.state == State.IO or proc.state == State.NOT_ARRIVED:
                proc.state = State.READY
            self.__ready_queue.append(proc)
        if proc.next_arrival_time == self.__timer and proc.state == State.IO_TERMINATED:
            proc.state = State.TERMINATED
            proc.turn_around_time = self.__timer - proc.arrival_time
            proc.terminate_time = self.__timer
            self.__done_list.append(proc)
    if self.__algorithm == Algorithm.SPN or self.__algorithm == Algorithm.SRT:
        self.__ready_queue.sort(key=lambda x: x.burst_time)
```

این متود به این صورت عمل می‌کند که هر بار صدا زده می‌شود پروسه‌هایی که وضعیت Ready دارند و در صف ready نیستند را به صف اضافه میکند حین این عملیات ملاحظاتی را نیز در نظر میگیرد که هر پروسه دو بار اضاف نشود یا اگر باید به این صف اضاف شود state را به ready تغییر دهد و از این قبیل ملاحظات.
البته این متود برای دو الگوریتم SRT و SPN صف را به صورت صعودی بر اساس burst time مرتب می‌کند.

الگوریتم‌ها:

fcfs:

```
def FCFS(self):
    self.update_ready_queue()
    while len(self.__done_list) != len(self.__processes):
        if self.__ready_queue:
            self.__running_process = self.__ready_queue.pop(0)
            self.__running_process.state = State.CPU
            if self.__running_process.response_time == -1:
                self.__running_process.response_time = self.__timer - self.__running_process.arrival_time
            while self.__running_process.state == State.CPU:
                self.__running_process.minus_burst_time()
                if self.__running_process.state == State.IO:
                    self.__running_process.next_arrival_time = self.__timer + self.__running_process.io_time + 1
                self.__timer += 1
                self.increment_waiting_time()
                if self.__running_process.state == State.TERMINATED:
                    self.__running_process.turn_around_time = self.__timer - self.__running_process.arrival_time
                    self.__running_process.terminate_time = self.__timer
                    self.__done_list.append(self.__running_process)
                self.update_ready_queue()
        else:
            self.__timer += 1
            self.__idle += 1
            self.update_ready_queue()
```

ابتدا صف Ready را به روز می‌کنیم و تا زمانی که همه پروسه‌ها به لیست پروسه‌های تمام شده اضاف نشده باشند مراحلی را انجام می‌دهد به این صورت که اگر صف Ready خالی نبود که به داخل شرط می‌رود اگر صف خالی بود به تایمر اضافه شده و همچنین idle هم اضاف می‌شود و دوباره صف Ready را آپدیت می‌کنیم .
حال به توضیح اینکه اگر صف Ready خالی نباشد می‌پردازیم:

از اول صف اولین پروسه را برمیداریم و شروع به انجام پردازش می‌کنیم و همینجا زمان response time را set می‌کنیم و تا زمانی که این اولین burst time این پروسه تمام نشده باشد cpu را در اختیار دارد و به ازای هر بار افزایش تایمر یک بار minus burst time صدا زده می‌شود برای پروسه در حال اجرا.
همچنین نیز به ازای هر بار تایمر صف Ready را به روز می‌کنیم و اگر پروسه به اتمام رسید وضعیت پروسه به Terminate تغییر می‌کند و ملاحظاتی که در تصویر مشخص است تنظیم می‌شود و به سراغ پروسه بعدی می‌رویم.

rr:

```
def RR(self):
    self.update_ready_queue()
    while len(self.__done_list) != len(self.__processes):
        if self.__ready_queue:
            self.__running_process = self.__ready_queue.pop(0)
            self.__running_process.state = State.CPU
            if self.__running_process.response_time == -1:
                self.__running_process.response_time = self.__timer - self.__running_process.arrival_time
            counter = 0
            while self.__running_process.state == State.CPU:
                self.__running_process.minus_burst_time()
                if self.__running_process.state == State.IO:
                    self.__running_process.next_arrival_time = self.__timer + self.__running_process.io_time + 1
                counter += 1
                self.__timer += 1
                self.increment_waiting_time()
                if self.__running_process.state == State.TERMINATED:
                    self.__running_process.turn_around_time = self.__timer - self.__running_process.arrival_time
                    self.__running_process.terminate_time = self.__timer
                    self.__done_list.append(self.__running_process)
                if counter == 5 and (self.__running_process.state not in [State.IO, State.IO_TERMINATED, State.TERMINATED]):
                    self.__running_process.state = State.READY
                    self.__running_process.next_arrival_time = self.__timer
                self.update_ready_queue()
        else:
            self.__timer += 1
            self.__idle += 1
            self.update_ready_queue()
```

کد الگوریتم‌ها مقدار زیادی شبیه هم می‌باشند که برای کم کردن مطلب فقط تفاوت‌ها توضیح داده می‌شود در ادامه.
در این الگوریتم به ازای time quantum مشخص شده که اینجا ۵ است پروسه به بیرون می‌رود و جای خود را به پروسه بعد می‌دهد شرطی که اضافه شده است برای ملاحظاتی است که پروسه اگر به io رفته یا قرار است با io تمام شود به مشکل برنخوریم.

spn:

```
def SPN(self):
    self.update_ready_queue()
    while len(self.__done_list) != len(self.__processes):
        if self.__ready_queue:
            self.__running_process = self.__ready_queue.pop(0)
            self.__running_process.state = State.CPU
            if self.__running_process.response_time == -1:
                self.__running_process.response_time = self.__timer - self.__running_process.arrival_time
            while self.__running_process.state == State.CPU:
                self.__running_process.minus_burst_time()
                if self.__running_process.state == State.IO:
                    self.__running_process.next_arrival_time = self.__timer + self.__running_process.io_time + 1
                self.__timer += 1
                self.increment_waiting_time()
                if self.__running_process.state == State.TERMINATED:
                    self.__running_process.turn_around_time = self.__timer - self.__running_process.arrival_time
                    self.__running_process.terminate_time = self.__timer
                    self.__done_list.append(self.__running_process)
                self.update_ready_queue()
        else:
            self.__timer += 1
            self.__idle += 1
            self.update_ready_queue()
```

این الگوریتم کاملا شبیه fcfs نوشته شده است ولی باید ذکر کرد که در متود update ready queue شرطی گذاشته شده بود که تفاوت این دو الگوریتم را در آنجا اعمال می‌کرد.

srt:

```
def SRT(self):
    self.update_ready_queue()
    while len(self.__done_list) != len(self.__processes):
        if self.__ready_queue:
            self.__running_process = self.__ready_queue.pop(0)
            self.__running_process.state = State.CPU
            if self.__running_process.response_time == -1:
                self.__running_process.response_time = self.__timer - self.__running_process.arrival_time
            while self.__running_process.state == State.CPU:
                self.__running_process.minus_burst_time()
                if self.__running_process.state == State.IO:
                    self.__running_process.next_arrival_time = self.__timer + self.__running_process.io_time + 1
                self.__timer += 1
                self.increment_waiting_time()
                if self.__running_process.state == State.TERMINATED:
                    self.__running_process.turn_around_time = self.__timer - self.__running_process.arrival_time
                    self.__running_process.terminate_time = self.__timer
                    self.__done_list.append(self.__running_process)
                self.update_ready_queue()
                if self.__ready_queue and self.__running_process.state == State.CPU:
                    if self.__running_process.burst_time > self.__ready_queue[0].burst_time:
                        self.__running_process.state = State.READY
                        self.__running_process.next_arrival_time = self.__timer
                        self.update_ready_queue()
                        break
        else:
            self.__timer += 1
            self.__idle += 1
            self.update_ready_queue()
```

این الگوریتم نیز همانند کد الگوریتم fcfs نوشته شده اما نکته‌ای که رعایت شده به این صورت است که هر بار که تایمر افزایش پیدا میکند و پروسه در حال اجرا یک میلی ثاینه پردازش شده است صف را آپدیت می‌کند و اگر که پروسه اول صف دارای burst time کمتر از پروسه در حال اجرا دارد، پروسه داخل cpu را بیرون می‌کند و متود update ready queue را صدا میزند که باعث می‌شود پروسه اول صف وارد cpu شده و مراحل به همینگونه ادامه پیدا کند در این حین باید دقت کند که پروسه‌ای هم داخل صف باشد که بخواهد آن را به داخل بیاورد.

main.py :

از این فایل برای ورودی دادن و instance گرفتن از Scheduler و پردازش کردن پروسه‌هایی که به زمانبند داده می‌شود استفاده می‌شود که در زیر به تابع‌های آن میپردازیم:

۱.

```
def csv_parser(file_path):
    processes = []
    with open(file_path, 'r') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        read_on = False
        for row in csv_reader:
            if read_on:
                processes.append(Process(int(row[0]), int(row[1]), [int(x) for x in row[2][1:-1].split(', ')], [int(x) for x in row[3][1:-1].split(', ')]))
            read_on = True
    return processes
```

این تابع آدرس فایل csv. را به عنوان ورودی می‌گیرد مقادیر آن که به صورت تصویر زیر می‌باشد را در یک لیست به نام processes ریخته و به عنوان خروجی برمی‌گرداند.

Process_id	Arrival time	CPU burst list	IO list
1	0	[6, 2]	[8]
2	1	[6, 2]	[8]
3	2	[6, 2]	[8]

۲. تابع اصلی که از کلاس Scheduler چهار تا instance میگیرد و به صورت multi-thread هر کدام را با یکی از الگوریتم‌های موجود اجرا می‌کند.

```
def main():
    file_path = "/Users/YahidGh/Documents/Operating System/CPU-Scheduler/processes.csv"
    processes = csv_parser(file_path=file_path)
    processes_fcfs = copy.deepcopy(processes)
    processes_rr = copy.deepcopy(processes)
    processes_spn = copy.deepcopy(processes)
    processes_srt = copy.deepcopy(processes)
    scheduler_fcfs = Scheduler()
    scheduler_rr = Scheduler()
    scheduler_spn = Scheduler()
    scheduler_srt = Scheduler()
    scheduler_fcfs.processes = processes_fcfs
    scheduler_rr.processes = processes_rr
    scheduler_spn.processes = processes_spn
    scheduler_srt.processes = processes_srt
    th_fcfs = threading.Thread(target=scheduler_fcfs.run(algorithm=Algorithm.FCFS))
    th_rr = threading.Thread(target=scheduler_rr.run(algorithm=Algorithm.RR))
    th_spn = threading.Thread(target=scheduler_spn.run(algorithm=Algorithm.SPN))
    th_srt = threading.Thread(target=scheduler_srt.run(algorithm=Algorithm.SRT))
    th_fcfs.start()
    th_rr.start()
    th_spn.start()
    th_srt.start()
    th_fcfs.join()
    th_rr.join()
    th_spn.join()
    th_srt.join()
    print('scheduling finished :)')
```