عنوان پروژه

cpu scheduler simulator

درس شبیه سازی

دکتر صفایی

پاییز ۱۴۰۱

گردآورندگان :

پرهام عسکرزاده ۹۸۱۷۰۹۳۵

حسام اثنی عشری ۹۸۱۷۰۶۳۵

فهرست

مقدمه

در این پروژه ما باید رفتار یک cpu scheduler را بررسی کنیم که دارای دو لایه است که در لایه دوم سه الگوریتم برای انجام پردازه های تولید شده در شرایط خاص قرار دارند

این پروژه به زبان پایتون نوشته شده و در ادامه به بررسی و معرفی بخش های کد و در آخر به اجرا کردن و آنالیز نتایج بدست آمده از آن میپردازیم

معرفی

ساختار کلی پروژه به صورت زیر است که در ادامه به بررسی کامل ها متد و کاربرد آن و همچنین نحوه کارکرد آن ها میپردازیم

همچنین لازم به ذکر است که بمنظور شبیه سازی زمان از لایبرری simpy استفاده کرده ایم

```
class Scheduler(object):
    def __init__(self, env, task_count, y_mean, x_rate, z_mean, k, quantum1, quantum2, duration):...

    def job_creator(self, x_rate, y_mean, z_mean):...

    def run(self):...

    def check_timeout(self):...

    def dispatcher(self):...

    def job_loader(self, k):...

    def __update_service_time_in_tuple(self, main_tuple, service_time):...

    def round_robin_t1_process(self, quantum_time):...

    def round_robin_t2_process(self, quantum_time):...

    def first_come_first_serve_process(self):...

    def __sort_priority_queue(self):...

    def analyse(self):...
```

مقادیر مورد نیاز و اولیه ما چه برای کارکرد سیستم و چه برای بررسی سیستم به صورت زیر است

```
def __init__(self, env, task_count, y_mean, x_rate, z_mean, k, quantum1, quantum2, duration):
    self.count = task_count
    self.y_mean = y_mean
    self.x_rate = x_rate
    self.z_mean = z_mean
    self.k = k
    self.quantum1 = quantum1
    self.quantum2 = quantum2
    self.env = env
    self.priority_queue = []
    self.round_robin_t1 = []
    self.round_robin_t2 = []
    self.first_come_first_serve = []
    self.priority_queue_count = []
    self.round_robin_t1_count = []
    self.round_robin_t2_count = []
    self.cpu_work_count = []
    self.waiting_time = []
    self.expired_processes = 0
    self.first_come_first_serve_count = []
    self.action = env.process(self.run())
    self.idle_status = True
    self.duration = duration
```

لایه اول و مقدمات

ابتدا لایبرری های مورد نیاز را نصب و به پروژه اضافه میکنیم

```
from numpy import random, arange
import simpy
import operator
import matplotlib.pyplot as plt
```

برای شروع ما ابتدا نیاز داریم تا تسک ها را با پارامتر های مورد نظر را ایجاد کنیم

کد هر تسک را ایجاد کرده و هر تسک به صورت تاپلی در می آید که اطلاعات زیر را به ترتیب دارا می باشد

(زمان ورود ، زمان مورد نیاز برای سرویس ، اولویت ، حداکثر زمان انتظار)

```
def job_creator(self, x_rate, y_mean, z_mean):
    # (enter arrival time, service time, priority, timeout)
    task_generated = (random.poisson(lam=x_rate, size=1)[0], int(random.exponential(scale=y_mean, size=1)[0]),
                      random.choice(arange(1, 4), p=[0.7, 0.2, 0.1], size=1)[0],
                      int(random.exponential(scale=z_mean, size=1)[0]))
    self.priority_queue.append(task_generated)
```

در مرحله بعد نیاز که متدی به عنوان job loader تعریف کنیم که وظیفه اصلی آن انتقال تسک ها از صف priority به صف های سرویس دهنده است این متد ابتدا بررسی میکند که کمتر از k تسک در صف های سرویس دهنده باشند و سپس k تسک را به آنها انتقال می دهد

```
def job_loader(self, k):
    self.__sort_priority_queue()
    self.priority_queue_count.extend(
        [len(self.priority_queue)] * (self.env.now - len(self.priority_queue_count) - 1))
    if len(self.round_robin_t1 + self.round_robin_t2 + self.first_come_first_serve) < k:
        counter = 0
        for idx, task in enumerate(self.priority_queue):
            if counter == k:
                break
            if task[0] <= self.env.now:
                self.round_robin_t1.append(self.priority_queue.pop(idx))
                counter += 1
```

وظیفه پخش کردن تسک های بین صف های سرویس دهنده بر عهده متدی به نام dispatcher است

لازم به توجه است که انتقال این تسک ها به صف به صورت تصادفی با احتمالات داده شده در داکیومنت اولیه پروژه (بخش امتیازی) است

```
def dispatcher(self):
    process = random.choice([self.round_robin_t1_process(self.quantum1), self.round_robin_t2_process(self.quantum2),
                             self.first_come_first_serve_process()], p=[0.8, 0.1, 0.1], size=1)[0]
    yield self.env.process(process)
```

همانطور که مشاهده میکنیم متدی به نام __sort_priority_queue داریم که وظیفه این متد مرتب کردن تسک ها در صف priority بر حسب اولویت و میزان زمان سرویس آنها است

```
def __sort_priority_queue(self):
    self.priority_queue = sorted(self.priority_queue, key=operator.itemgetter(2, 1), reverse=True)
```

هر پردازه ما بعد از هر واحد زمانی جلو رفتن نیاز دارد که ماکزیمم timeout آن چک شود که اگر از زمان آن گذشته است آن تسک از سامانه خارج شود (بخش امتیازی)

```
def check_timeout(self):
    for queue in [self.round_robin_t1, self.round_robin_t2, self.priority_queue, self.first_come_first_serve]:
        for task in queue:
            if task[3] < self.env.now:
                queue.remove(task)
                self.expired_processes += 1
```

لایه دوم

حال به بررسی الگوریتم های اجرای تسک ها توسط پردازنده میپردازیم :

Round Robin type1 :

در این الگوریتم ما یک کوانتوم تایمی داریم که به پردازه اختصاص میدهیم اگر پردازه در این کوانتوم تایم به اتمام رسید به تسک بعدی میپردازیم اما اگر این تسک در کوانتوم تایم مشخص شده تمام نشد آن تسک را به همراه سرویس تایم جدید خود به صف بعدی منتقل می کنیم

در این الگوریتم لازم است که بخشی از اطلاعات مورد نیاز که در آخر پروژه برای آنالیز مورد کاربرد هست کامل میشود مانند طول صف مربوطه ، delay ها و ..

```
def round_robin_t1_process(self, quantum_time):
    if len(self.round_robin_t1) == 0:
        self.round_robin_t1_count.extend([0] * (self.env.now - len(self.round_robin_t1_count)))
        self.idle_status = True
    for task in self.round_robin_t1:
        if task[1] <= quantum_time:
            self.round_robin_t1_count.extend(
                [len(self.round_robin_t1)] * (self.env.now - len(self.round_robin_t1_count) - 1))
            self.waiting_time.append((self.env.now + task[1]) - task[0])
            self.round_robin_t1.remove(task)
            self.round_robin_t1_count.append(len(self.round_robin_t1))
            self.idle_status = False
            yield self.env.timeout(task[1])
        else:
            self.round_robin_t1_count.extend(
                [len(self.round_robin_t1)] * (self.env.now - len(self.round_robin_t1_count) - 1))
            self.round_robin_t2.append(self.__update_service_time_in_tuple(task, task[1] - quantum_time))
            self.round_robin_t1.remove(task)
            self.round_robin_t1_count.append(len(self.round_robin_t1))
            self.idle_status = False
            yield self.env.timeout(quantum_time)
```

Round Robin type2 :

مانند الگوریتم اول است تنها با این تفاوت که کوانتوم تایم متفاوتی دارد و اگر تسکی در کوانتوم تایم مورد نظر به اتمام نرسید به صف لایه بعد یعنی First Come First Serve منتقل میشود

```
def round_robin_t2_process(self, quantum_time):
    if len(self.round_robin_t2) == 0:
        self.round_robin_t2_count.extend([0] * (self.env.now - len(self.round_robin_t2_count)))
        self.idle_status = True
    for task in self.round_robin_t2:
        if task[1] <= quantum_time:
            self.round_robin_t2_count.extend(
                [len(self.round_robin_t2)] * (self.env.now - len(self.round_robin_t2_count) - 1))
            self.waiting_time.append((self.env.now + task[1]) - task[0])
            self.round_robin_t2.remove(task)
            self.round_robin_t2_count.append(len(self.round_robin_t2))
            self.idle_status = False
            yield self.env.timeout(task[1])
        else:
            self.round_robin_t2_count.extend(
                [len(self.round_robin_t2)] * (self.env.now - len(self.round_robin_t2_count) - 1))
            self.first_come_first_serve.append(self.__update_service_time_in_tuple(task, task[1] - quantum_time))
            self.round_robin_t2.remove(task)
            self.round_robin_t2_count.append(len(self.round_robin_t2))
            self.idle_status = False
            yield self.env.timeout(quantum_time)
```

: First Come First Serve

این الگوریتم به صورت FIFO عمل میکند

در این الگوریتم نیز مانند الگوریتم های قبلی اطلاعات لازم برای آنالیز سامانه در آخر شبیه سازی پر میشوند

```
def first_come_first_serve_process(self):
    if len(self.first_come_first_serve) > 0:
        self.first_come_first_serve_count.extend(
            [len(self.first_come_first_serve)] * (self.env.now - len(self.first_come_first_serve_count) - 1))
        task = self.first_come_first_serve.pop()
        self.waiting_time.append((self.env.now + task[1]) - task[0])
        self.first_come_first_serve_count.append(len(self.first_come_first_serve_count))
        self.idle_status = False
        yield self.env.timeout(task[1])
    else:
        self.first_come_first_serve_count.extend([0] * (self.env.now - len(self.first_come_first_serve_count)))
        self.idle_status = True
```

همانطور که دیدیم در هرمرحله اجرا ، تسک های خارج نشده از سیستم سرویس تایمشان بعد از هر نوع اجرا آپدیت میشود برای این کار از متد زیر استفاده میکنیم

```
def __update_service_time_in_tuple(self, main_tuple, service_time):
    new_tuple = list(main_tuple)
    new_tuple[1] = service_time
    return tuple(new_tuple)
```

در آخر ما نیاز به متدی داریم که بتوانیم مراحل اجرا را مرتب و پشت هم انجام دهیم و به عبارتی سامانه با اجرای آن شروع به شبیه سازی می شود و ما با استفاده از آن میتوانیم مدیریت سامانه را انجام دهیم (تعداد تسک ها و ..)

```
def run(self):
    time = 0
    period_time = 2
    for _ in range(self.count):
        self.job_creator(y_mean=self.y_mean, x_rate=self.x_rate, z_mean=self.z_mean)
    while True:
        self.check_timeout()
        if time % period_time == 0 or time == 0:
            self.job_loader(k=self.k)
        time = self.env.now
        yield self.env.process(self.dispatcher())
        if self.idle_status:
            self.cpu_work_count.append(1)
        if time == self.env.now:
            yield self.env.timeout(1)
        if time + self.env.now > time + period_time:
            time += period_time
```

همانطور که در بخش اول پروژه دیدیم ما متدی به نام analyse داریم که به بررسی و انالیز سامانه اعمم از میزان زمان نتظار تسک ها در صف ، میزان بهره وری cpu ، طول صف ها در هر الگوریتم و همچنین رسم پلات هایی که دید شهودی و بهتری به ما میدهند می پردازد

```
def analyse(self):
    queues = {
        'priority queue': self.priority_queue_count,
        'round robin t1 queue': self.round_robin_t1_count,
        'round robin t2 queue': self.round_robin_t2_count,
        'first come first serve queue': self.first_come_first_serve_count,
    }
    for i in queues:
        plt.bar(list(range(len(queues[i]))), queues[i])
        plt.xlabel('time')
        plt.ylabel('count')
        plt.title(i)
        plt.show()
        try:
            print('mean', i, 'delay =', sum(queues[i]) / len(queues[i]))
        except ZeroDivisionError:
            print('mean', i, 'delay =', 0)
    print('Percentage of expired processes =', self.expired_processes / self.count * 100)
    print('waiting time mean =', sum(self.waiting_time) / len(self.waiting_time))
    print('cpu worked time=', 100 - (len(self.cpu_work_count) / self.duration) * 100)
```

اجرا

حال نیاز است در این مرحله پروژه را با ورودی های دلخواه اجرا گرفته و به بررسی آنالیز و آمار سامانه شبیه سازی خود بپردازیم و سپس به سوالات مطرح شده پاسخ دهیم

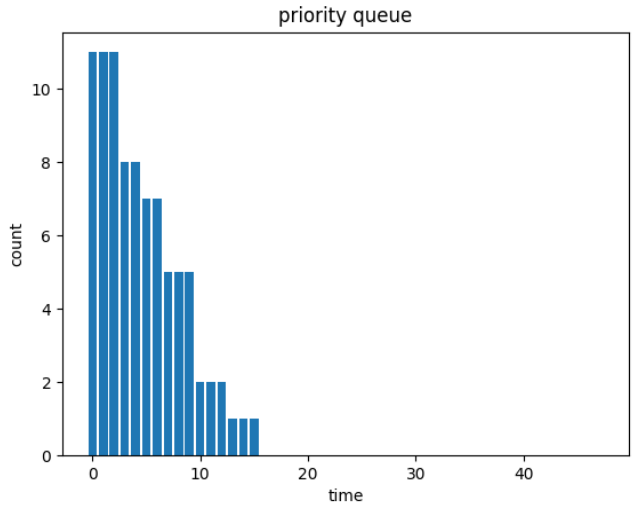
ورودی :

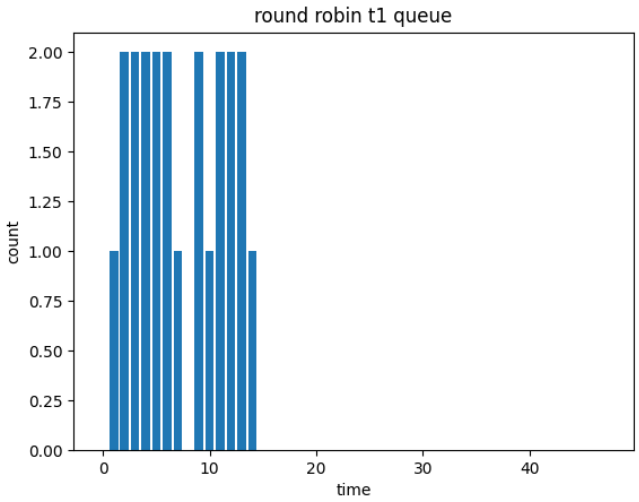
هنگامی که پروژه را با ورودی های زیر مقدار میدهیم و به مدت ۵۰ واحد زمانی اجرا میکنیم به آمار عددی زیر میرسیم

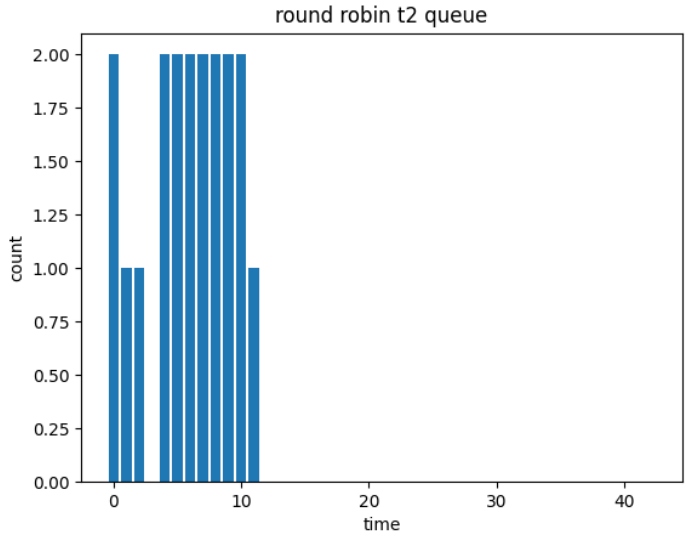
```
if __name__ == '__main__':
    env = simpy.Environment()
    duration = 50
    simulation = Scheduler(env=env, task_count=25, y_mean=2, z_mean=10, x_rate=3, k=2, quantum1=1, quantum2=2,
                           duration=duration)
    env.run(until=duration)
    simulation.analyse()
```

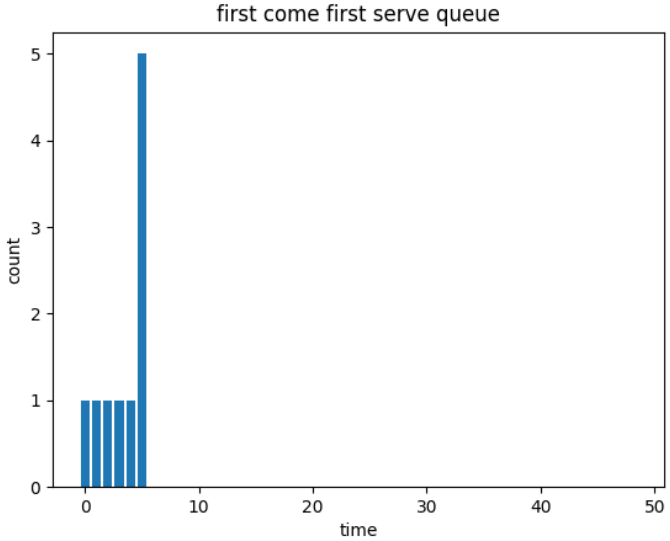
```
heduler › analyse() › for i in queues › except ZeroDivisionError
main ×
mean round robin t1 queue delay = 0.4583333333333333
mean round robin t2 queue delay = 0.4418604651162791
mean first come first serve queue delay = 0.20408163265306123
Percentage of expired processes = 72.0
waiting time mean = 10.428571428571429
cpu worked time= 32.0

Process finished with exit code 0
```

همچنین نمودار های مربوط به طول صف ها و به طور کلی تعداد تسک ها در صف ها در ادامه آمده است




round robin t1 queue
count
time
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
10
20
30
40


round robin t2 queue
count
time
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
10
20
30
40


first come first serve queue
count
time
0
1
2
3
4
5
0
10
20
30
40
50

خروجی ها :

۱ - (میانگین طول صف ها ، میانگین زمان صرف شده در صف ها ، میزان بهره وری cpu ، درصد پردازه های منقضی شده) :

پاسخ این سوال ها در عکس مربوط به نتایج عددی آنالیز در بالاتر (مربوط به بخش اجرا) آمده است

۲ – پیشنهاد خود را برای بهبود میانگین زمان صرف شده در صف ها بنویسید.

همانطور که از نمودار ها بر می آید ما بیشترین میزان صف را در الگوریتم ها ران رابین داریم برای بهبود شرایط میتوانیم چند کار از جمله کار های زیر انجام دهیم :

۱- تسک های موجود در صف را بر حسب سرویس تایم آنها مرتب کنیم اگر توانستیم آن را در کوآنتوم تایم تعریف شده انجام دهیم آن را نگه میداریم و در غیر این صورت آن را به لایه پایینی یا حتی لایه FCFS انتقال میدهیم

۲- میتوانیم کوانتوم تایم ها را به روشی تغییر دهیم که ابتدا تسک هایی که نیاز به سرویس تایم کمی دارند در لایه اول و آنها که سرویس تایم متوسطی دارند در لایه دوم و آنان که سرویس تایم بیشتری دارند در لایه بعدی قرار گیرند و همانطور که مشخص است کوآنتوم تایم مورد نظر با توجه به تخمین اندازه سرویس تایم های تسک ها بدست می آوریم