بسمه تعالی

سیستم‌های بی‌درنگ

گزارش پروژه فاز دوم

دکتر صفری

مهدی علیزاده ۹۹۱۰۱۹۳۲

مهدی قائم‌پناه ۹۹۱۰۹۱۹۹

منتور: خانم ملکی

دانشگاه صنعتی شریف

بهار ۱۴۰۲

مقدمه

در این گزارش، ما دو پروتکل مهم اشتراک منابع در سیستم‌های چندپردازشی، یعنی MSRP و MrsP را مورد بررسی قرار داده و شبیه‌سازی انجام شده برای ارزیابی این پروتکل‌ها را ارائه می‌دهیم. این شبیه‌سازی‌ها شامل تولید وظایف مصنوعی، تخصیص منابع به وظایف و ارزیابی زمان‌بندی وظایف با استفاده از الگوریتم EDF می‌باشد. همچنین برای نگاشت وظایف به پردازنده‌ها از الگوریتم WFD استفاده شده است.

۱ بررسی پروتکل‌ها

پروتکل MSRP:

پروتکل MSRP برای مدیریت انسداد وظایف در سیستم‌های چندپردازشی طراحی شده است. در این پروتکل، زمانی که یک وظیفه به دلیل دسترسی به یک منبع دچار انسداد می‌شود، سایر وظایف در همان پردازنده نیز متوقف می‌شوند. این امر باعث می‌شود که وظیفه انسداد شده بدون هیچ‌گونه تداخل زمانی که منبع مورد نیازش آزاد شد، به اجرای خود ادامه دهد.

پروتکل MRSP:

پروتکل MrsP برای بهبود عملکرد پروتکل MSRP طراحی شده است. در این پروتکل، زمانی که یک وظیفه به دلیل دسترسی به یک منبع دچار انسداد می‌شود، سایر وظایف در همان پردازنده می‌توانند به اجرای خود ادامه دهند. این امر باعث افزایش بهره‌وری پردازنده‌ها می‌شود.

۲ بررسی الگوریتم‌های زمان‌بندی

الگوریتم EDF:

الگوریتم EDF یکی از معروف‌ترین الگوریتم‌های زمان‌بندی در سیستم‌های بلادرنگ است. در این الگوریتم، وظیفه‌ای که نزدیک‌ترین مهلت (deadline) را دارد، برای اجرا انتخاب می‌شود. این الگوریتم سعی می‌کند وظایف را به گونه‌ای زمان‌بندی کند که هیچ وظیفه‌ای مهلت خود را از دست ندهد.

الگوریتم WFD:

الگوریتم WFD برای نگاشت وظایف به پردازنده‌ها استفاده می‌شود. در این الگوریتم، وظایف براساس زمان اجرای بدترین حالت (WCET) مرتب شده و سپس به پردازنده‌ای تخصیص داده می‌شوند که کمترین استفاده را دارد. این الگوریتم سعی می‌کند بار پردازنده‌ها را به طور یکنواخت توزیع کند.

۳ گزارش کار

در ابتدا اشتباهاتی که در فاز اول داشتیم رو درست می‌کنیم. در فاز اول سطوح قبضگی و همچنین مشخص شدن پردازنده به صورت استاتیک پیاده‌سازی نشده بود که در این فاز ابتدا به سراغ این نکات می‌رویم.

```python
def generate_tasks(n, U_total, min_period, max_period, num_resources=2, num_critical_sections=2, max_preemption_level=3):
    utilizations = uunifast(n, U_total)
    tasks = []
    for i, u in enumerate(utilizations):
        period = random.randint(min_period, max_period)
        wcet = int(u * period) + 1
        deadline = period
        number_of_critical_sections = random.randint(1, num_critical_sections)
        preemtaion_level = random.randint(1, max_preemption_level)
        criticality = []
        for _ in range(number_of_critical_sections):
            critical_section_length = random.randint(1, 2)
            critical_section_start = random.randint(0, wcet-1)
            critical_section_end = random.randint(critical_section_start, min(wcet, critical_section_start + critical_section_length))
            critical_resource = random.randint(0, num_resources - 1)
            criticality.append({
                'start': critical_section_start, 'end': critical_section_end, 'resource': critical_resource
            })
        tasks.append({
            'id': i,
            'period': period,
            'wcet': wcet,
            'start_time': None,
            'deadline': deadline,
            'preemption_level': preemtaion_level,
            'criticality': criticality,
            'remaining_time': wcet,
            'arrival': 0,
            'utilization': u,
            'blocking': False,
        })
    return tasks
```

در فاز اول تسک‌ها را با استفاده از الگوریتم uunifast تولید کردیم و در این فاز همانطور که می‌بینید سطح قبضگی و همچنین مشکل کردن تعداد و زمان نواحی بحرانی را به کمک کد بالا در تسک‌ها مشخص می‌کنیم. توجه کنید که کانفیگ‌هایی که برای نواحی بحرانی درنظر گرفته شده اند را می‌توان به مقادیر دلخواه تغییر داد تا بتوان شبیه‌سازی را در فازهای مختلفی اجرا کرد.

سپس به سراغ نگاشات این تسک‌ها به پردازنده‌ها می‌رویم که این کار را با استفاده از الگوریتم wfd انجام می‌دهیم که کد آن را می‌توانید در زیر مشاهده کنید. توضیحات الگوریتم wfd نیز در بالا و همچنین گزارش فاز اول داده شده است.

```python
def wfd_mapping(tasks, num_processors):
    processors = [[] for _ in range(num_processors)]

    for task in sorted(tasks, key=lambda t: -t['utilization']):
        worst_fit_processor = min(processors, key=lambda p: sum(t['utilization'] for t in p))
        worst_fit_processor.append(task)
        task['processor'] = processors.index(worst_fit_processor)
    return processors
```

حال به سراغ پیاده‌سازی الگوریتم‌های زمان‌بندی می‌رویم. برای پیاده‌سازی ابتدا نیاز به تابعی داریم تا با استفاده از ان ceilingها را مصاحبه کنیم. این مقادیر در هنگام تصمیم‌گیری برای قبضه کردن یا نکردن یک تسک به کار می‌رود. این مقادیر را به سادگی با استفاده از تعاریف گفته شده در کلاس و تابع زیر بدست می‌آوریم:

```python
def calculate_ceilings(tasks, num_resources):
    ceilings = []
    for r in range(num_resources):
        max_preemption_level = 0
        for t in tasks:
            for c in t['criticality']:
                if c['resource'] == r:
                    max_preemption_level = max(max_preemption_level, t['preemption_level'])
        ceilings.append(max_preemption_level)
    return ceilings
```

حال به سراغ پیاده‌سازی خود الگوریتم MSRP می‌رویم.

در این الگوریتم یک آرایه از تسک‌های حال حاضر بر روی هر پردازنده نگه می‌داریم و سپس در ابتدای هر واحد زمانی چک می‌کنیم که اگر کار هر تسک با ریسورس‌های آن تمام شده باشد آن ریسورس‌ها را آزاد می‌کنیم تا در اختیار بقیه‌ی تسک‌ها بتوانند قرار بگیرند.

```
def simulate_msrp(processors, num_resources, end_time, ceilings):
    num_schedulable = 0
    num_non_scheduleable = 0
    time = 0
    resources_status = num_resources * [None]
    pi = 0
    current_task = len(processors) * [None]
    while time < end_time:
        print(10*'-')
        print('Time: {}'.format(time))
        for i, p in enumerate(processors):
            if current_task[i] is None:
                continue
            for c in current_task[i]['criticality']:
                if c['end'] < current_task[i]['wcet'] - current_task[i]['remaining_time']:
                    if resources_status[c['resource']] == current_task[i]['id']:
                        resources_status[c['resource']] = None
```

حال همانطور که گفته شده بود با استفاده از الگوریتم EDF بر روی هر پردازنده مشخص می‌کنیم که چه تسکی الویت دارد که اجرا شود.

```
for i, p in enumerate(processors):
    task = edf_schedule(p, time)

    if task is None:
        continue
    if current_task[i] is None:
        current_task[i] = task
    elif task['id'] != current_task[i]['id']:
        if task['preemption_level'] > pi:
            current_task[i] = task
    if current_task[i] is None:
        continue
```

حال لازم است که چک کنیم آیا این تسک می‌تواند اجرا شود یا خیر. برای این کار چک می‌کنیم که آیا تسک در زمان حال حاضر نیاز به ریسورس ناحیه بحرانی دارد یا خیر و اگر بله آیا آن ریسورس آزاد هست یا خیر. اگر جواب بله بود تمام ریسورس‌ها را در اختیار می‌گیرد و شروع به اجرا می‌کند و در غیر این صورت به حالت بلاک می‌رود و در آن پردازنده کاری انجام نمی‌شود تا ریسورس‌ها آزاد شوند و در اختیار تسک قرار گیرند.

```
for c in current_task[i]['criticality']:
    if c['end'] < current_task[i]['wcet'] - current_task[i]['remaining_time']:
        if resources_status[c['resource']] == current_task[i]['id']:
            resources_status[c['resource']] = None
    if c['start'] <= current_task[i]['wcet'] - current_task[i]['remaining_time'] < c['end']:
        if resources_status[c['resource']] is not None and  resources_status[c['resource']] != current_task[i]['id']:
            current_task[i]['blocking'] = True
            blocked = True
            print('Task {} is blocking for resourse {}'.format(current_task[i]['id'], c['resource']))
if not blocked:
    current_task[i]['blocking'] = False
```

در ادامه نیز کد مربوط به مرحله‌ی اجرا شدن و همچنین پایان یافتن اجرا را مشاهده می‌کنید که در صورتی که تسک پایان یافت ریسورس‌هایی که در اختیار داشت را آزاد می‌کند و مقادیر برای تسک بعدی ست می‌شوند.

```
if current_task[i]['blocking'] is False:
    for c in current_task[i]['criticality']:
        if c['start'] <= current_task[i]['wcet'] - current_task[i]['remaining_time'] < c['end']:
            resources_status[c['resource']] = current_task[i]['id']
            print('Task {} is using resource {} on proceccor {}'.format(current_task[i]['id'], c['resource'], current_task[i]['processor']))
    current_task[i]['remaining_time'] -= 1
    print('Task {} is running on proceccor {}'.format(current_task[i]['id'], current_task[i]['processor']))
    if current_task[i]['remaining_time'] == 0:
        for c in current_task[i]['criticality']:
            if resources_status[c['resource']] == current_task[i]['id']:
                resources_status[c['resource']] = None
        current_task[i]['arrival'] += current_task[i]['period']
        current_task[i]['deadline'] += current_task[i]['period']
        current_task[i]['remaining_time'] = current_task[i]['wcet']
        current_task[i]['blocking'] = False
        current_task[i] = None
        num_schedulable += 1
pi = max([ceilings[r] for r in range(num_resources) if resources_status[r] is not None], default=0)
```

توجه کنید که ما مقدار pi را در پایان هر واحد زمانی آپدیت می‌کنیم و هر تسک تنها در صورتی می‌تواند تسک در حال اجرا را قبضه کند که سطح قبضگی آن از مقدار pi در آن لحظه بیشتر باشد.

حال به سراغ الگوریتم MRSP می‌رویم. توجه کنید که این الگوریتم کاملا مشابه الگوریتم بالا است تنها وقتی یک تسک به خاطر نبود ریسورس مورد نیازش بلاک می‌شود آن پردازنده به طور کلی به حالت انتظار نمی‌رود بلکه یک تسک دیگر را انتخاب می‌کند که بتواند آن را اجرا کند. برای این کار تابع EDF را اینگونه تغییر می‌دهیم که یک مجموعه از تسک‌ها را به ترتیب ددلاین برمی‌گرداند تا اگر یکی از تسک‌ها را نتوانستین اسکجول کنیم تسک دیگری را بتوانیم انتخاب کنیم:

```
def edf_schedule_order(tasks, time):
    ready_tasks = [t for t in tasks if t['arrival'] <= time and t['remaining_time'] > 0]
    if ready_tasks:
        sorted(ready_tasks, key= lambda t: t['deadline'])
        return ready_tasks
    return None
```

سپس از این تابع در انتخاب تسک در حلقه‌ی اصلی استفاده می‌کنیم.

```
for i, p in enumerate(processors):
    tasks = edf_schedule_order(p, time)

    if tasks is None:
        continue

    for task in tasks:
        if current_task[i] is not None and task['id'] != current_task[i]['id'] and task['preemption_level'] <= pi:
            continue


        blocked = False
        for c in task['criticality']:
            if c['start'] <= task['wcet'] - task['remaining_time'] < c['end']:
                if resources_status[c['resource']] is not None and  resources_status[c['resource']] != current_task[i]['id']:
                    blocked = True
                    break
        if blocked:
            continue
```

همانطور که مشاهده می‌کنید بر روی تسک‌های سورت شده توسط EDF یک for می‌زنیم و امتحان می‌کنیم هر تسک بلاک شده است یا نه. اگر بلاک نشده بود آن تسک را انتخاب کرده و با آن تسک scheduling را ادامه می‌دهیم.

در آخر هم برای رسم نمودار میانگین تسک‌های زمان‌بندی شده یک ماژول برای رسم نمودار نوشتیم که بعد از پایان اجرا نمودار را رسم کرده و در یک فایل ذخیره می‌کند.

```python
import matplotlib.pyplot as plt

def draw(avg, name):
    run = [i for i in range(1, 11)]

    plt.plot(avg, run)
    plt.xlabel("run")
    plt.ylabel("average schedule-ables")
    plt.title("number of proccesor = 2")
    plt.savefig(f'{name}.png')
```

۴ نتایج

در ابتدا می‌توانید لاگ اجرای هر مورد را در اجرای هر بار مشاهده کنید:

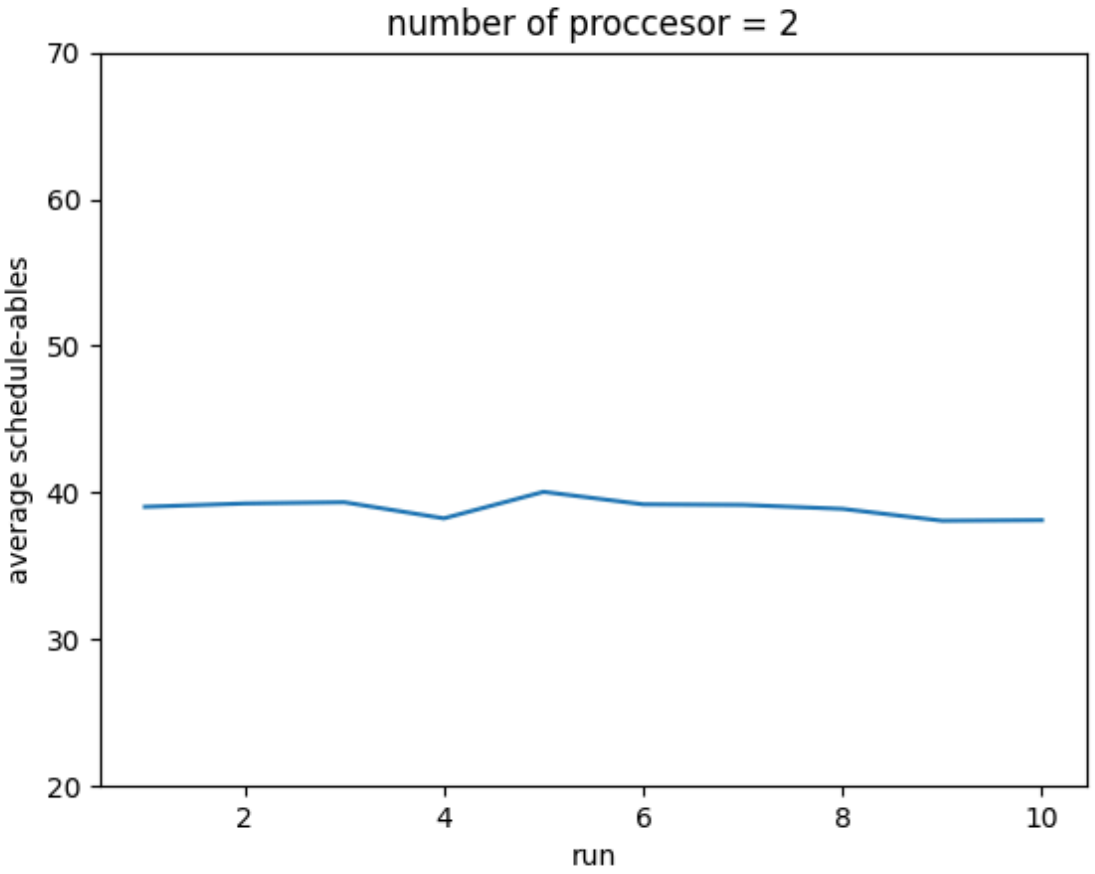
```
----------
Time: 181
Task 2 is using resource 1 on proceccor 0
Task 2 is running on proceccor 0
Task 0 is blocking for resourse 1 on proceccor 1
```

به عنوان مثال در لاگ بالا نمایش داده می‌شود که در زمان ۱۸۱ تسک ۲ ریسورس ۱ را در اختیار دارد و بر روی پردازنده با آیدی ۰ در حال اجرا است و همچنین تسک با آیدی ۰ بر روی ریسورس ۱ در پردازنده‌ی ۱ بلاک شده است.

حال در ادامه نمودارهای مربوط به هر کدام از کانفیگ‌های گفته شده در صورت پروژه را مشاهده می‌کنید. ابتدا نمودارها برای الگوریتم **MSRP** را رسم می‌کنیم:

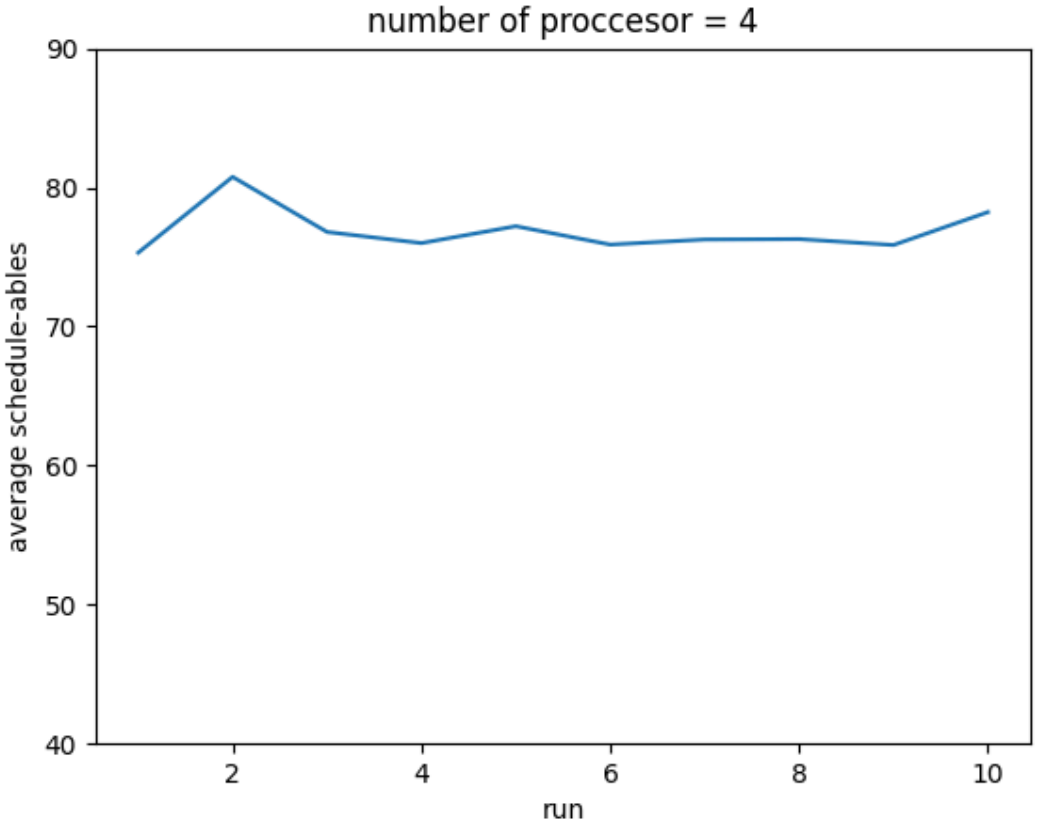
تعداد ریسورس‌ها برابر با تعداد پردازنده‌ها

۱. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۲۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت



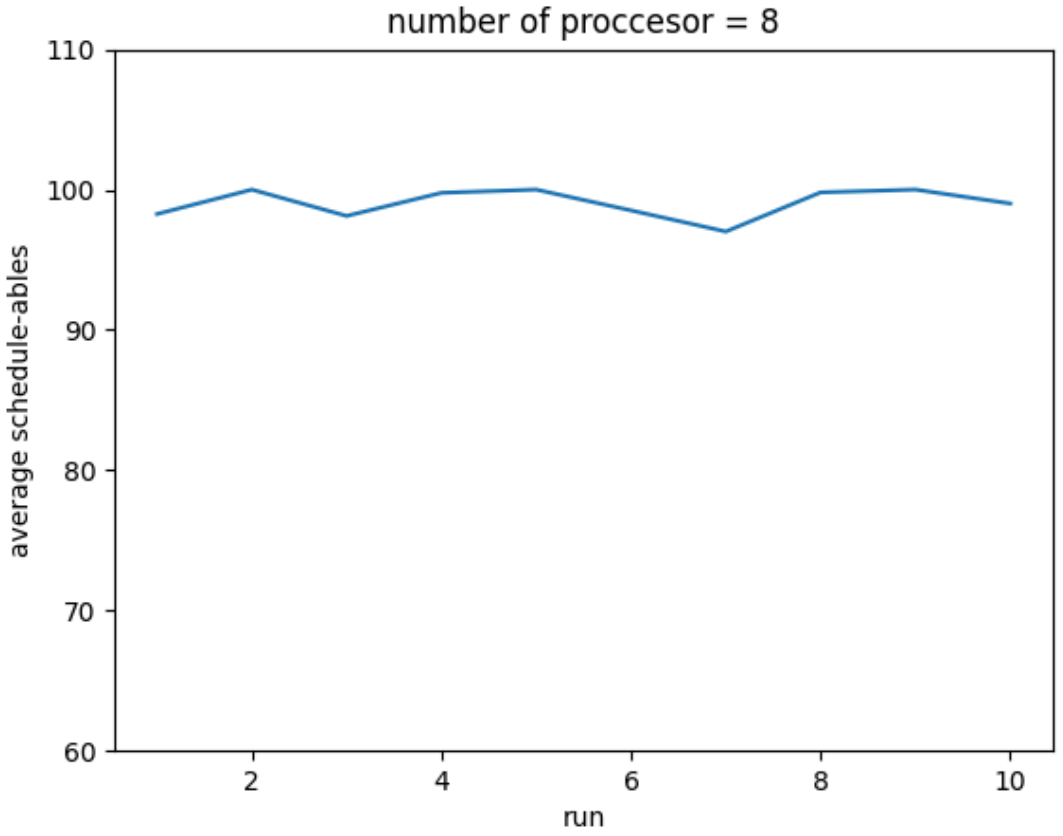
Output:

```
[39.03368216433961, 39.26002701089858, 39.34521690034447, 38.24830416581405,
40.05495539880716, 39.203233482697655, 39.1659225141389, 38.89332921064044,
38.084937026556624, 38.123776465049076]
```



Output:

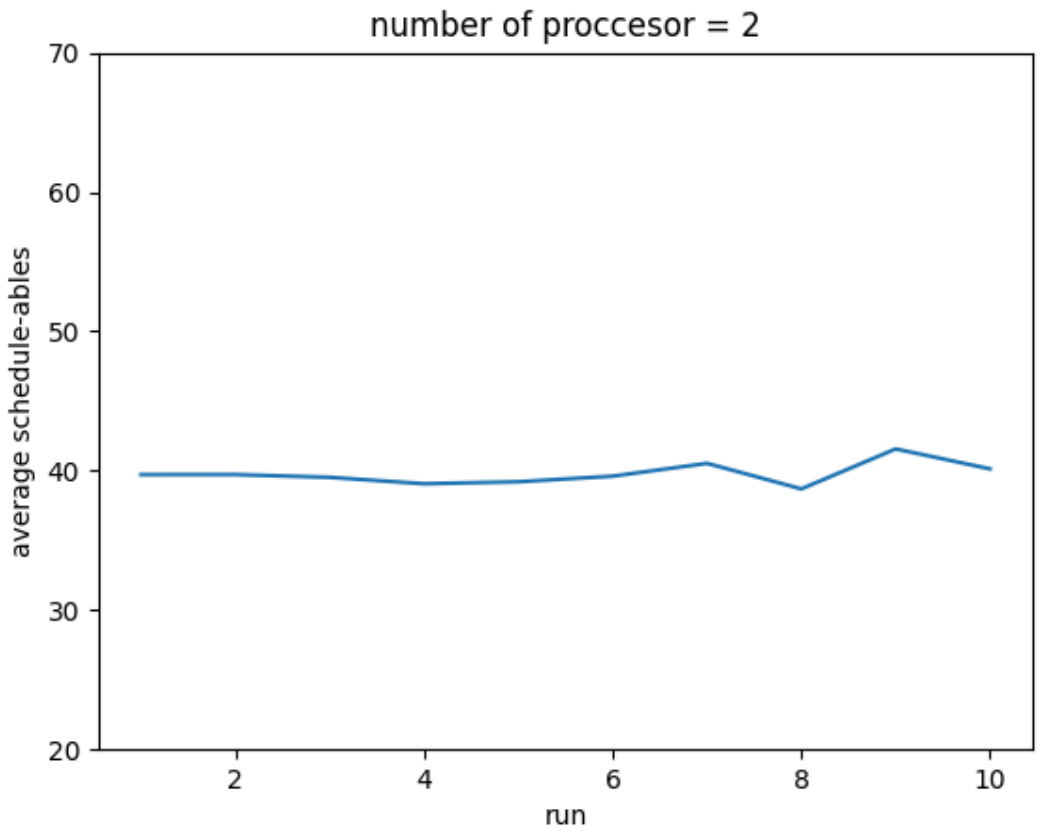
```
   [75.30760155064891, 80.75754142964155, 76.79660489081799, 75.99563785038738,
77.2094944515327, 75.88835550572122, 76.2542392591138, 76.2837816567768,
75.85857685516892, 78.22215885237202]
```



Output:

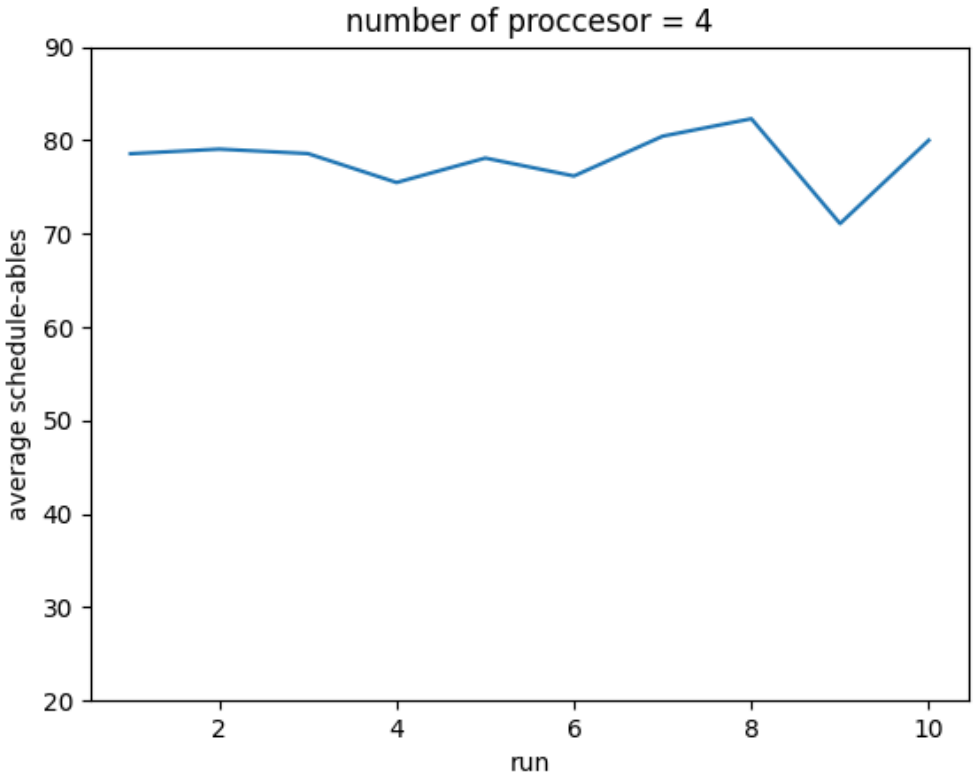
```
[98.26011314981174, 100.0, 98.12464649617283, 99.77077840772109, 100.0,
98.51635495673749, 97.01007337635104, 99.80255523590891, 100.0, 99.01007337635104]
```

۲. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت



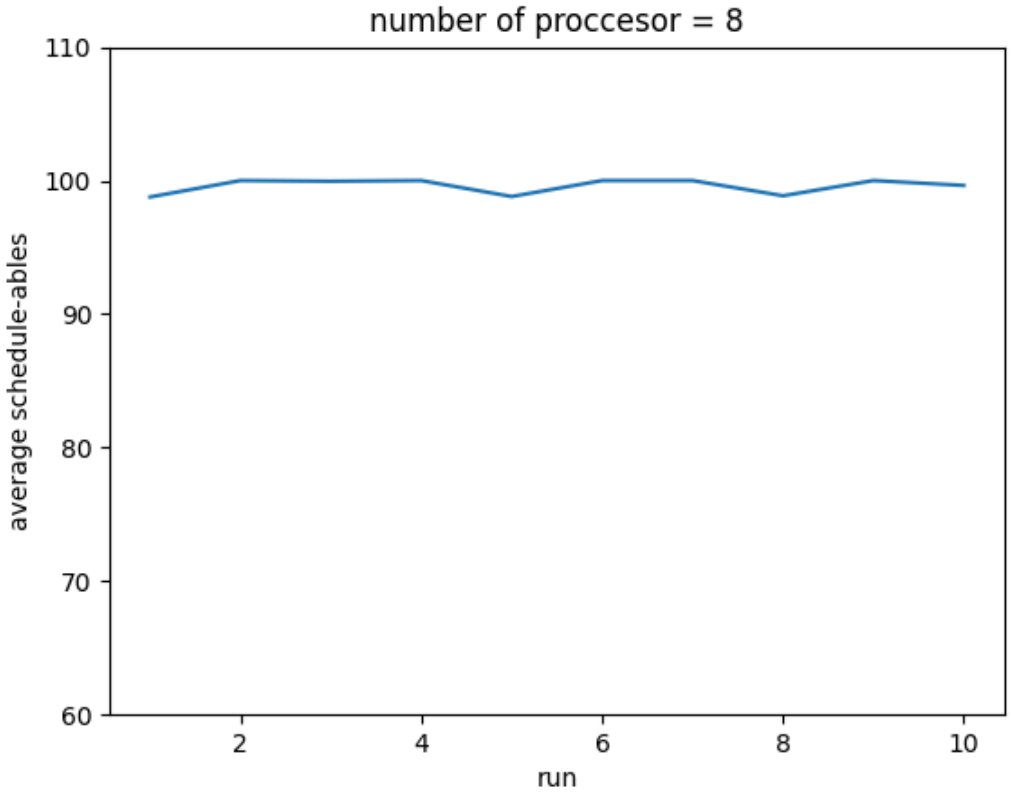
Output:

```
[39.71738814725924, 39.721866505942046, 39.5242918465012, 39.063184172171056,
39.20677419544084, 39.607193218912876, 40.52819787296322, 38.69570303174982,
41.562979002992705, 40.130248643595394]
```



Output:

```
[78.56076372607374, 79.05246492329529, 78.57476082040115, 75.47751349348094,
78.09397834885534, 76.17449787236427, 80.43832581188911, 82.30405271305521,
71.07895883988337, 80.00260217002393]
```



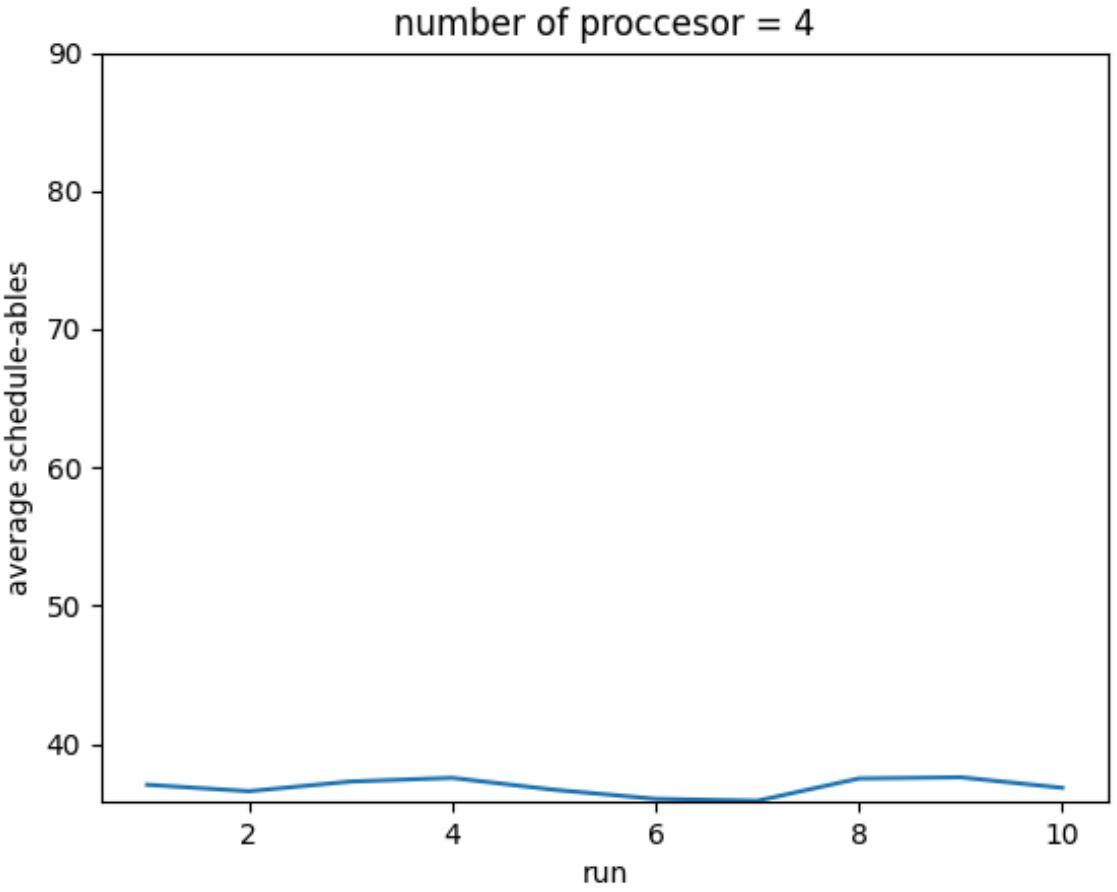
Output:

```
[98.76789488317652, 100.0, 99.94735795421312, 100.0, 98.80397103395705, 100.0,
100.0, 98.86063236181457, 100.0, 99.63507353096036]
```

تعداد ریسورس‌ها یک عدد رندوم بین ۲ تا ۶:

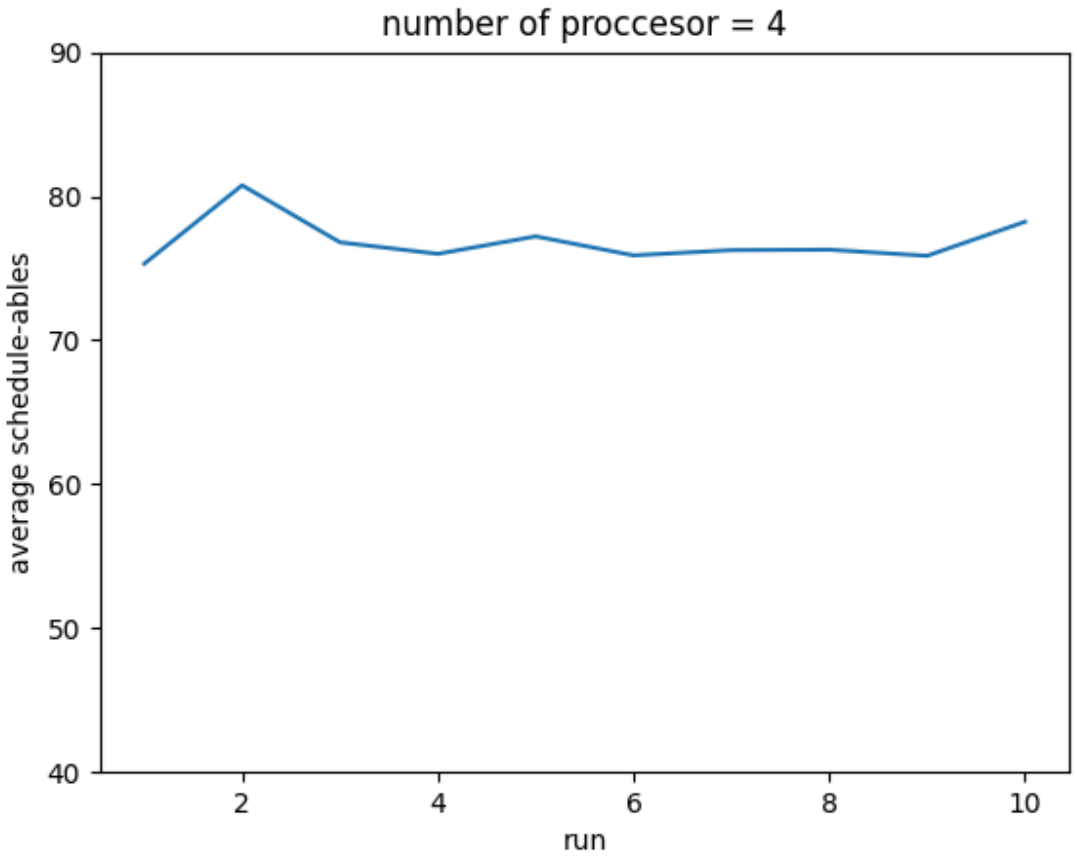
نکته: در این اجراها عدد ۴ به عنوان عدد رندوم انتخاب شد ولی شما می‌توانید با اجرای دوباره‌ی کد عددهای دیگر را هم تست کنید.

۱. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۲۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت:



Output:

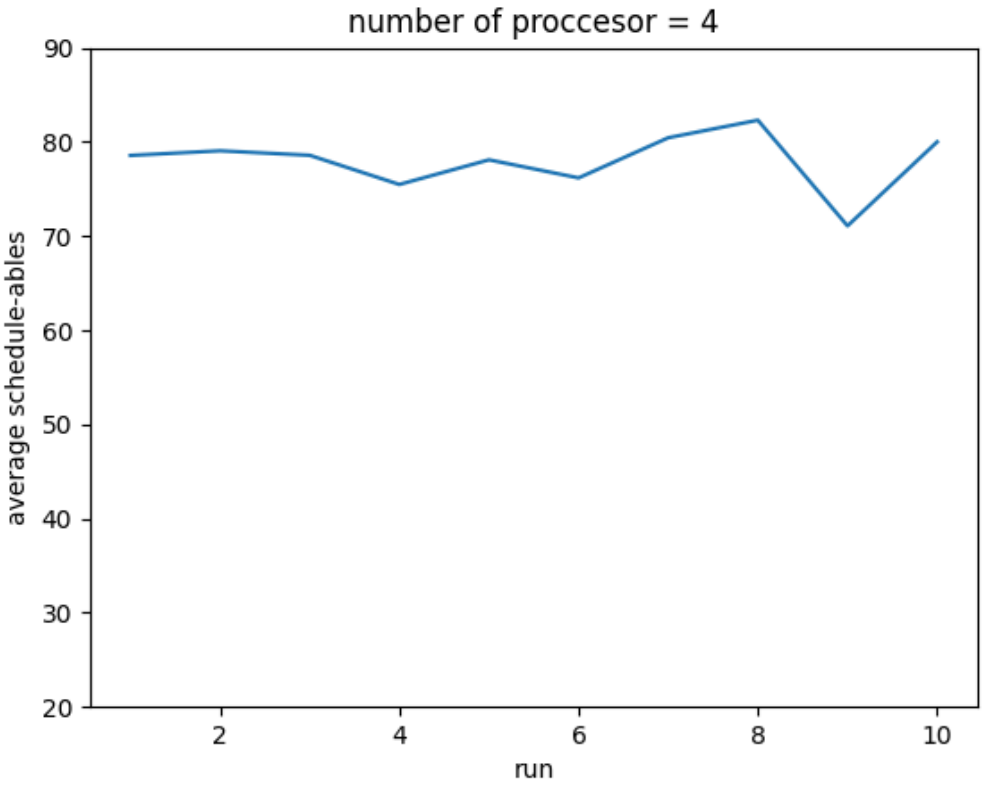
```
[37.040539696420765, 36.57325253522702, 37.272747266125876, 37.53741227636651,
36.678061223503114, 36.021078886134475, 35.87979226721921, 37.4920532115925,
37.58068225221999, 36.816862044872295]
```



Output:

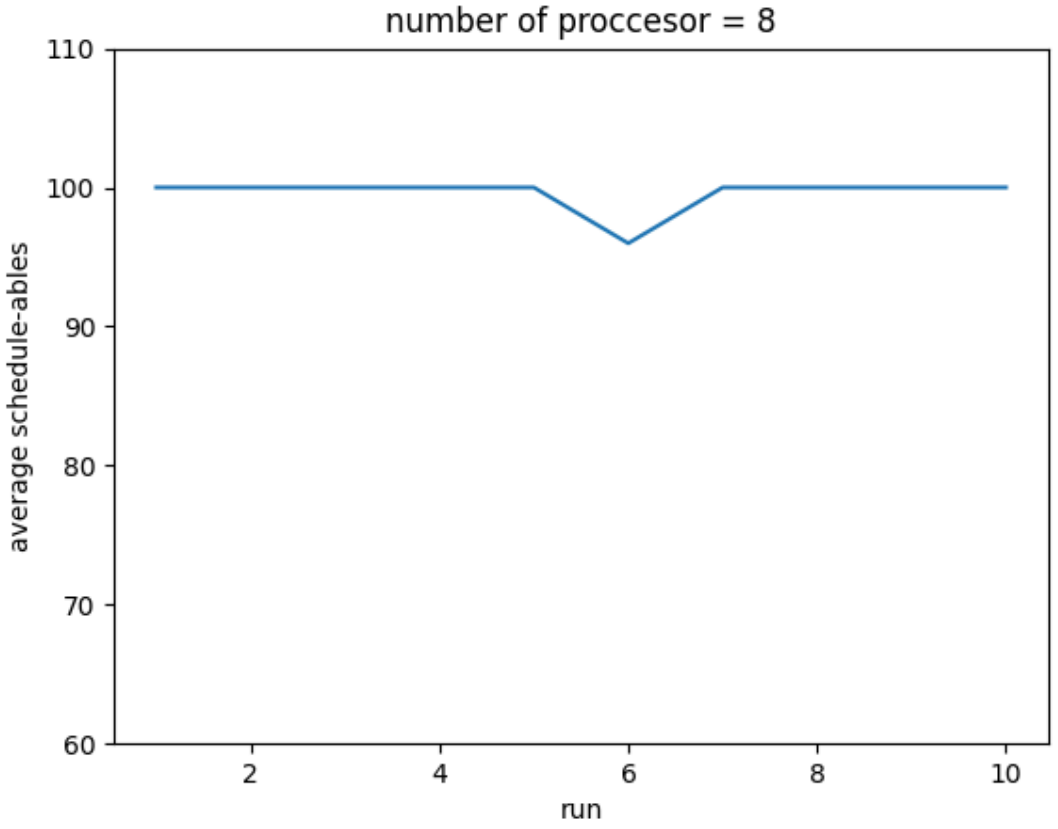
```
   [75.30760155064891, 80.75754142964155, 76.79660489081799, 75.99563785038738,
77.2094944515327, 75.88835550572122, 76.2542392591138, 76.2837816567768,
75.85857685516892, 78.22215885237202]
```

۲. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت:



Output:

```
[78.56076372607374, 79.05246492329529, 78.57476082040115, 75.47751349348094,
78.09397834885534, 76.17449787236427, 80.43832581188911, 82.30405271305521,
71.07895883988337, 80.00260217002393]
```



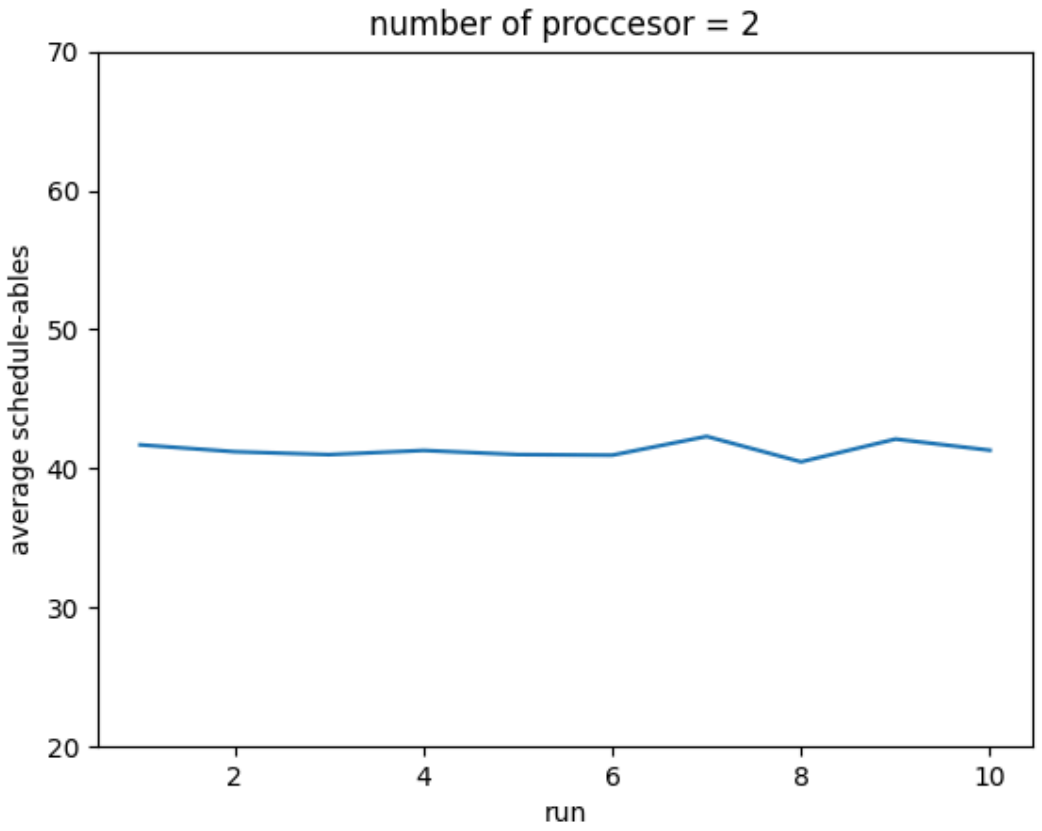
Output:

```
[100.0, 100.0, 100.0, 100.0, 100.0, 95.98250915172936, 100.0, 100.0, 100.0, 100.0]
```

حال تمام این نمودارها را برای الگوریتم **MRSP** نیز می‌کشیم تا بتوانیم نتایج را بررسی کنیم و مقایسه کنیم.

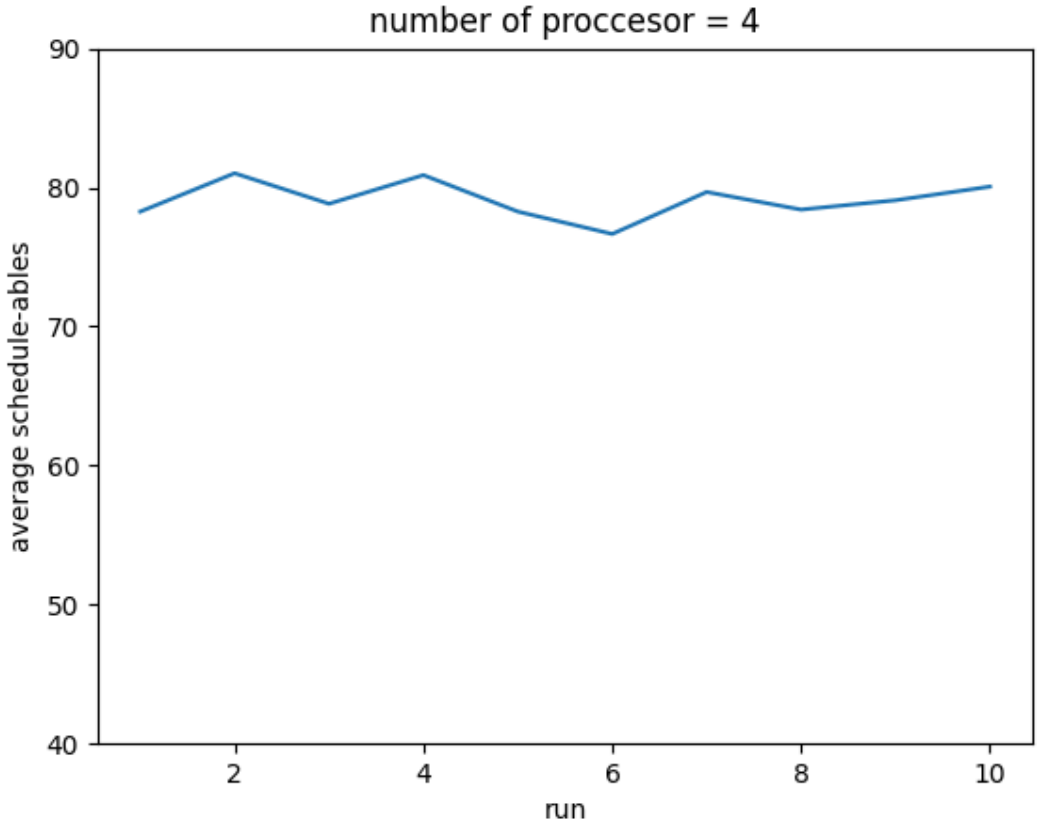
تعداد ریسورس‌ها برابر با تعداد پردازنده‌ها

۱. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۲۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت



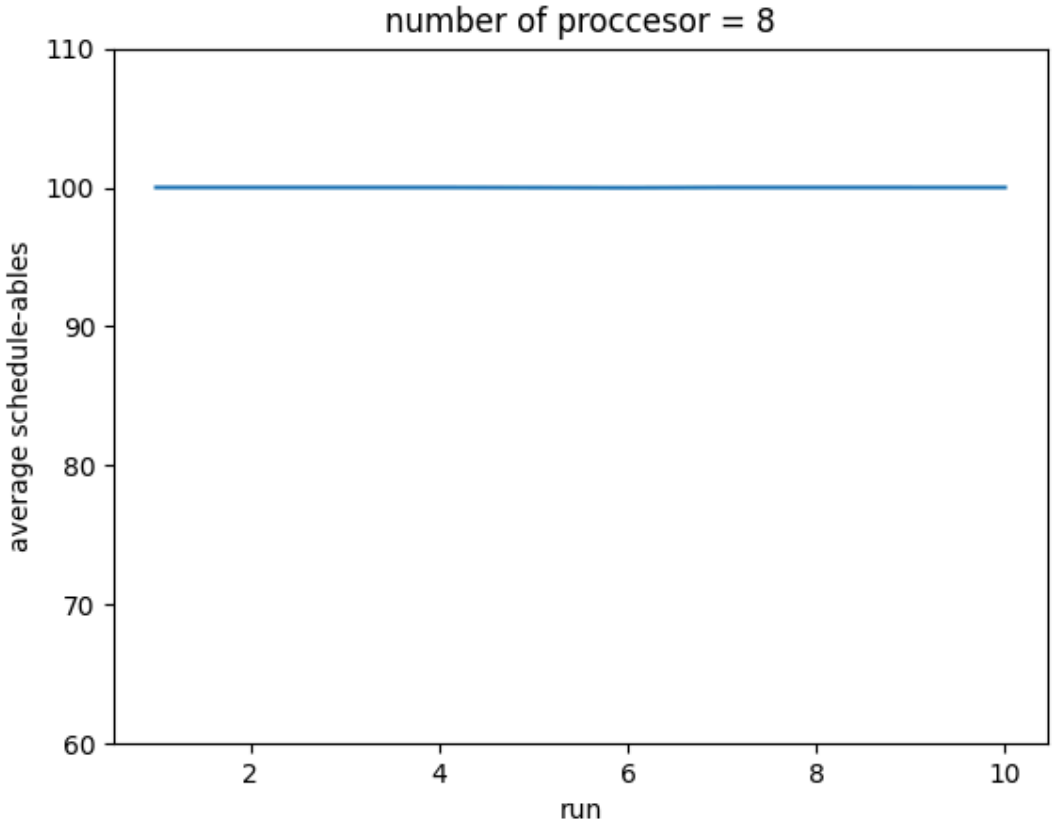
output:

```
[41.703573123063215, 41.211150082637495, 41.00064741074702, 41.30045195519749,
41.004602543100084, 40.96235457301155, 42.309658798893096, 40.48314822939035,
42.118429604191576, 41.31698533083277]
```



Output:

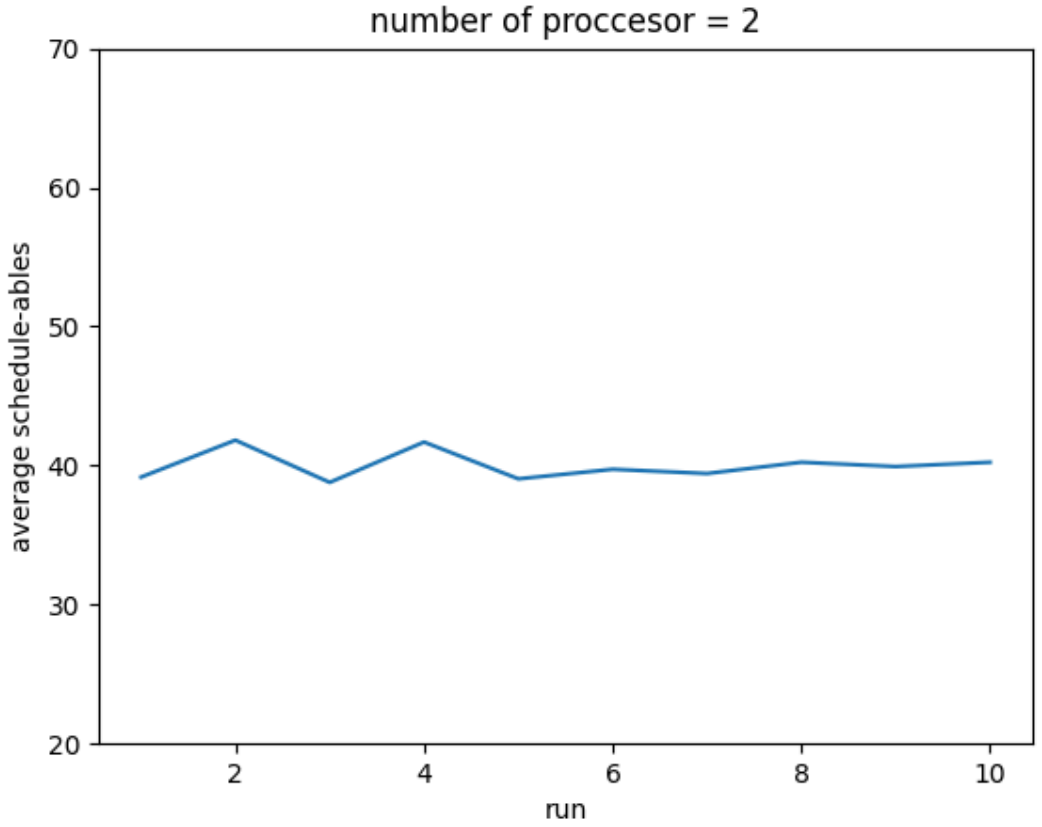
```
[78.25933540070183, 81.01833430856252, 78.82522466924891, 80.88479694034558,
78.26236146883221, 76.65197217574519, 79.67094746123325, 78.40436853329572,
79.07528575817493, 80.05650132338249]
```



Output:

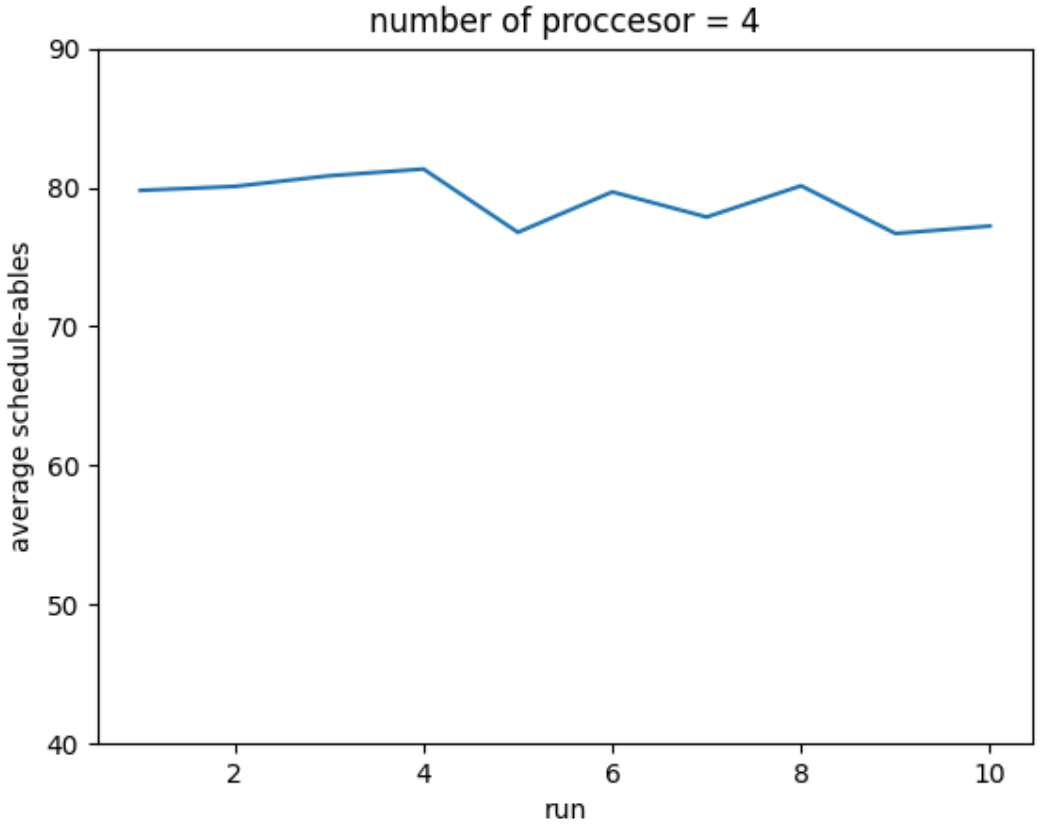
```
[100.0, 100.0, 100.0, 100.0, 99.99072011878248, 99.97944521854278,
99.99980452447646, 99.99980700532086, 99.9988043068866, 99.99783444531285]
```

۲. میانگین زمان‌بندپذیری ۴۰۰ وظیفه در ۱۰ بار اجرا با بهره‌وری ۰.۵ به ازای هر پردازنده و تعداد پردازنده‌های متفاوت



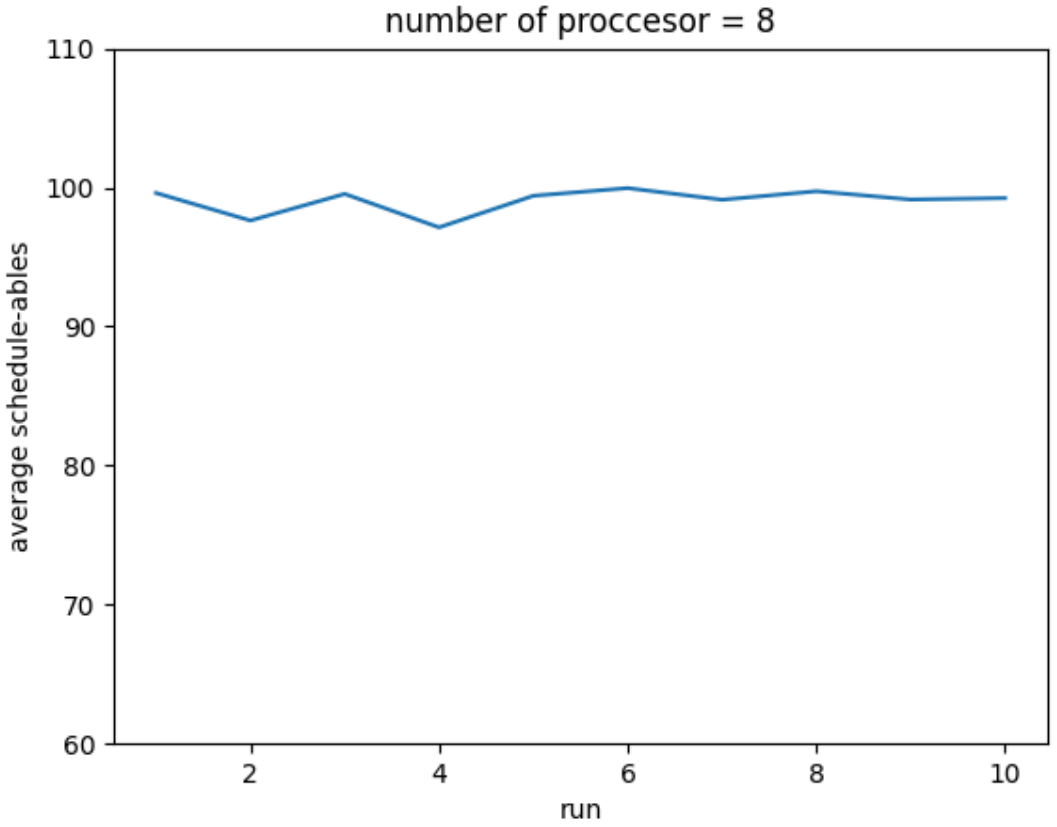
Output:

```
[39.167798305427056, 41.82445052895598, 38.787345607084596, 41.68977612285211,
39.04306370763469, 39.72986322560432, 39.426970160226865, 40.237543458681564,
39.92814506647373, 40.23641804342353]
```



Output:

```
[79.7847804601188, 80.07029210543374, 80.83517126480096, 81.32231552563994,
76.77162911608428, 79.67652347261853, 77.87113145228312, 80.11635220125787,
76.68449117449742, 77.22238227114873]
```



Output:

```
[99.59842255946532, 97.61910188642646, 99.53110887318847, 97.12672896982887,
99.39833442005794, 99.95090493691116, 99.11381818323554, 99.72509902424886,
99.12872602522499, 99.23255396346849]
```

در نهایت می‌توان نتیجه گرفت که الگوریتم MRSP با توجه به اینکه از زمان‌های خالی بهتر استفاده می‌کند به طور کلی پرفورمنس بهتری نسبت به الگوریتم MSRP دارد. همچنین توجه کنید که به طور کلی به افزایش Utilization نیز میانگین تعداد تسک‌های زمان‌بندپذیر بیشتر می‌شود. همچنین با افزایش تعداد ریسورس‌ها و ثابت ماندن پردازنده‌ها میانگین مورد نظر کاهش میابد. چون احتمال تداخل و بلاک شدن بیشتر می‌شود.