فاز اول پروژه

دنیا روشن‌ضمیر، ابوالفضل سلطانی

تولید وظیفه‌ها

در این بخش، یک موجودیت برای هر گره ایجاد کرده‌ایم. هر گره یک `Job` است. با توجه به سطح بحرانی وظیفه‌اش، یک یا دو بدترین زمان اجرا دارد که با تابعی رندم، عددی بین 1 تا `MAX_EXECUTION_TIME` انتخاب می‌شود.

```
class Job:
   def __init__(self, id: int, critical: bool):
       self.id = id
       self.critical = critical
       self.wcet_low = rand.randint(1, MAX_EXECUTION_TIME)
       if critical:
           self.wcet_high = rand.randint(self.wcet_low, MAX_EXECUTION_TIME)
```

هر وظیفه یک `Task` است؛ که با `id` یکتا دارد، برای deadline و period آن مقدار ذخیره می‌شود. همچنین لیستی از زیربخش‌ها که باید اجرا شوند، و یال‌های مربوط به زیربخش‌ها یا گره‌ها. ایجاد یک وظیفه با متد `generate_task` ساخته می‌شود. به طور تصادفی بین ۵ تا ۱۰ زیربخش برای یک وظیفه در نظر گرفته شده است. مقدار period عددی رندم بین حداقل زمان مورد نیاز برای اجرای تمامی زیربخش‌ها و `MAX_PERIOD` می‌باشد. مقدار deadline نیز عددی رندم بین حداقل زمان مورد نیاز برای اجرای تمامی زیربخش‌ها و period است.

نکته: مقاله مربوط به FFT را تا ساعت ۶ نتوانستیم درست کنیم. گراف وابستگی وظایف فعلا به صورت یک درخت دودویی درامده. بعد از پیدا کردن دقیق FFT آن را آپدیت خواهیم کرد.

```
class Task:
   def __init__(self, id: int, deadline: int, period: int, jobs: list[Job],
dependency_graph: list[tuple[int, int]]):
       self.id = id
       self.deadline = deadline
```

```
        self.period = period
        self.jobs = jobs
        self.dependency_graph = dependency_graph

    @staticmethod
    def generate_task(id: int, critical: bool):
        num_jobs = rand.randint(5, 10)
        jobs: list[Job] = []
        for i in range(num_jobs):
            j = Job(i, critical)
            jobs.append(j)
        if critical:
            sum = sum([j.wcet_high for j in jobs])
        else:
            sum = sum([j.wcet_low for j in jobs])
        period = rand.randint(sum, MAX_PERIOD)
        deadline = rand.randint(sum, period)
        dependency_graph = Task._generate_dependency_graph(num_jobs)
        return Task(id, deadline, period, jobs, dependency_graph)

    @staticmethod
    def _generate_dependency_graph(num_jobs: int):
        edges: tuple[int, int] = []
        for i in range(2, num_jobs // 2 + 1):
            edges.append((i//2, i))
        for i in range(num_jobs // 2 + 1, num_jobs):
            edges.append((i, i + 1))
        label = 2 ** mlg
        for j in range(mlg):
            for k in range(2 ** (mlg)):
                edges.append((label + k, (2 ** mlg) + label + k))
                edges.append((label + k, ((label + k) ^ (2 ** j)) + (2**mlg)))
            label += 2 ** mlg

        nodes = [i + 1 for i in range(label + 2 ** mlg - 1)]
        return (nodes, edges)
```

با استفاده از تابع show گراف را نمایش می‌دهیم.

```
def show(t: Task):
```

```
    g = networkx.DiGraph()
    g.add_nodes_from([job.id for job in t.jobs])
    for src, sink in t.dependency_graph:
        g.add_edge(src, sink)

    levels = np.zeros(len(t.jobs) + 1, dtype=int)
    for src, sink in t.dependency_graph:
        if levels[sink] == 0:
            levels[sink] = levels[src] + 1



    max_level = int(np.max(levels))
    levels_width = np.zeros(max_level + 1, dtype=int)
    for level in levels:
        levels_width[level] += 1
    levels_width[0] = 1

    max_width = max(levels_width)
    layout = networkx.spring_layout(g)
    x_array = np.zeros(max_level + 1, dtype=int)
    for i in range(len(t.jobs)):
        level = levels[i + 1]
        x: int | None = None
        x = (x_array[level] - ((levels_width[level] - 1) / 2)) * (max_width /
levels_width[level]) * 10

        layout[i + 1] = (x, -10 * level)
        x_array[level] += 1

    node_attributes = {
        'node_color': 'pink',
        'node_size': 400,
        'font_size': 12,
        'font_color': 'black',
    }
    edge_attributes = {
        'edge_color': 'gray',
        'width': 1,
        'arrows': True,
        'arrowstyle': '-|>',
        'arrowsize': 12,
```

```
    }
    plt.figure(figsize=(6, 6))
    networkx.draw_networkx(g, layout, with_labels=True, **node_attributes,
**edge_attributes)

    plt.title(f'Task {t.id}')

    plt.show()
```

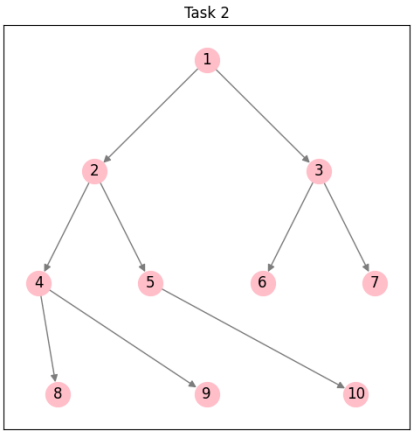
حال با نسبت 1 (احتمال ۱/۲) هر وظیفه را High Critical یا Low Critical قرار می‌دهیم و 5 وظیفه می‌سازیم.

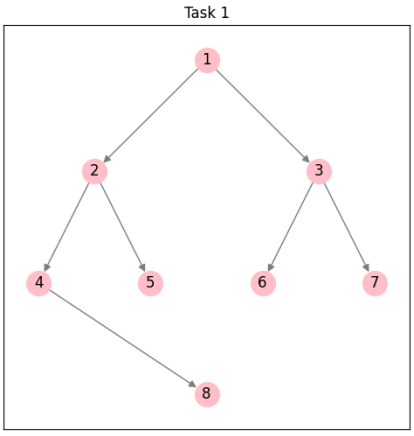
```
num_tasks = 5
tasks = []

ratio = 0.5

for i in range(num_tasks):
    is_task_critical = rand.random() < ratio
    tasks.append(Task.generate_task(i + 1, is_task_critical))
    show(tasks[i])
```

خروجی بعد از اجرا کردن این قطعه کد، ۵ وظیفه ساخته می‌شود و گراف‌ها آن‌ها نیز نشان داده می‌شود.





تولید منابع

```python
def generate_resources(num_nodes, num_resources):
    resources = {}
    for i in range(1, num_resources + 1):
        resources[f"Resource_{i}"] = random.randint( a: 1,  b: 16)
    return resources
```

`generate_resources(num_nodes, num_resources)`:

این تابع منابع را به صورت تصادفی ایجاد می‌کند و آن‌ها را به عنوان یک دیکشنری باز می‌گرداند.

```python
def assign_resources_to_critical_sections(resources, num_critical_sections):
    critical_sections = {}
    for i in range(1, num_critical_sections + 1):
        critical_sections[f"Critical_Section_{i}"] = {}
        for resource, value in resources.items():
            critical_sections[f"Critical_Section_{i}"][resource] = random.randint( a: 0, value)
    return critical_sections
```

`assign_resources_to_critical_sections(resources, num_critical_sections)`:

این تابع منابع را به بخش‌های بحرانی اختصاص می‌دهد و مقادیر تصادفی را به هر منبع در هر بخش بحرانی اختصاص می‌دهد. نتیجه این عملیات یک دیکشنری از بخش‌های بحرانی و تخصیص‌های آنها به منابع است.

```python
def main():
    num_nodes = random.randint( a: 1,  b: 5)
    num_critical_sections = random.randint( a: 1,  b: 16)
    num_resources = random.randint( a: 1,  b: 5)

    resources = generate_resources(num_nodes, num_resources)
    critical_sections = assign_resources_to_critical_sections(resources, num_critical_sections)

    print("Generated Resources:")
    for resource, value in resources.items():
        print(f"{resource}: {value}")

    print("\nAssigned Resources to Critical Sections:")
    for section, resource_allocation in critical_sections.items():
        print(f"{section}: {resource_allocation}")
```

`main()`: این تابع از دو تابع قبلی استفاده می‌کند و منابع را ایجاد کرده، به بخش‌های بحرانی اختصاص می‌دهد و نتایج را نمایش می‌دهد.

ورودی‌ها:

1. `num_nodes`: تعداد گره‌ها.
2. `num_resources`: تعداد منابع.
3. `num_critical_sections`: تعداد بخش‌های بحرانی.

که طبق داک داده شده مشخص میشود.

خروجی‌ها:

1. منابع تولید شده به صورت یک دیکشنری که نام هر منبع به همراه مقدار آن است.
2. اختصاص منابع به بخش‌های بحرانی به صورت یک دیکشنری که نام هر بخش بحرانی به همراه تخصیص منابع به آن است.

نمونه خروجی:

منابع و اختصاص آنها به بخش بحرانی به صورت رندم انجام میشود. بنابراین می‌توانید دو نمونه خروجی متفاوت را در زیر میتوانید در زیر ببینید:

```
Generated Resources:
Resource_1: 13
Resource_2: 7
Resource_3: 12

Assigned Resources to Critical Sections:
Critical_Section_1: {'Resource_1': 11, 'Resource_2': 5, 'Resource_3': 12}
Critical_Section_2: {'Resource_1': 0, 'Resource_2': 7, 'Resource_3': 1}
Critical_Section_3: {'Resource_1': 2, 'Resource_2': 3, 'Resource_3': 5}
Critical_Section_4: {'Resource_1': 12, 'Resource_2': 5, 'Resource_3': 6}
```

```
Generated Resources:
Resource_1: 3
Resource_2: 8
Resource_3: 8
Resource_4: 10

Assigned Resources to Critical Sections:
Critical_Section_1: {'Resource_1': 2, 'Resource_2': 1, 'Resource_3': 0, 'Resource_4': 2}
Critical_Section_2: {'Resource_1': 3, 'Resource_2': 4, 'Resource_3': 2, 'Resource_4': 9}
Critical_Section_3: {'Resource_1': 0, 'Resource_2': 4, 'Resource_3': 0, 'Resource_4': 4}
Critical_Section_4: {'Resource_1': 2, 'Resource_2': 5, 'Resource_3': 6, 'Resource_4': 2}
Critical_Section_5: {'Resource_1': 3, 'Resource_2': 4, 'Resource_3': 3, 'Resource_4': 0}
Critical_Section_6: {'Resource_1': 0, 'Resource_2': 8, 'Resource_3': 8, 'Resource_4': 8}
Critical_Section_7: {'Resource_1': 0, 'Resource_2': 0, 'Resource_3': 3, 'Resource_4': 5}
Critical_Section_8: {'Resource_1': 0, 'Resource_2': 8, 'Resource_3': 5, 'Resource_4': 10}
Critical_Section_9: {'Resource_1': 2, 'Resource_2': 6, 'Resource_3': 3, 'Resource_4': 6}
Critical_Section_10: {'Resource_1': 1, 'Resource_2': 2, 'Resource_3': 8, 'Resource_4': 0}
Critical_Section_11: {'Resource_1': 0, 'Resource_2': 2, 'Resource_3': 5, 'Resource_4': 4}
```

توضیحات:

- مدت زمان اجرای هر منبع برای هر گره به صورت تصادفی انتخاب می‌شود و نمی‌تواند بیشتر از مقدار آن منبع باشد.

نگاشت منابع:

الگوریتم شامل تعیین سقف زمانی و درخواست مهلت برای منابع مختلف است. به طور کلی، دو فرمول ارائه شده در داک برای این منظور به کار می‌روند.

توضیحات الگوریتم:

1. **Ψr(t)**: سقف زمانی برای منبع Rr در زمان t را تعیین می‌کند.
2. **Πr(t)**: درخواست مهلت برای منبع Rr در زمان t را مشخص می‌کند.

تابع تعیین سقف زمانی: این تابع بررسی می‌کند که آیا منبع Rr در زمان t توسط τi در دسترس است یا نه و بر اساس آن سقف زمانی را تعیین می‌کند.

```
def psi_r(t, R_r, tau_i, held_by_tau_i):
    if held_by_tau_i:
        return t + R_r['psi_ri']
    else:
        return math.inf
```

تابع تعیین درخواست مهلت: این تابع بررسی می‌کند که آیا منبع Rr توسط یک وظیفه در زمان t نگه داشته شده است یا نه و بر اساس آن زودترین مهلت را تعیین می‌کند.

```
def pi_r(t, R_r, jobs):
    if R_r['held_by_job']:
        earliest_deadline = min(job['deadline'] for job in jobs if job['needs_resource'] == R_r)
        return earliest_deadline
    else:
        return math.inf
```

محاسبه سقف زمانی و درخواست مهلت:

در این بخش، برای هر منبع موجود در `resources`، سقف زمانی و درخواست مهلت محاسبه می‌شود و نتایج چاپ می‌شود. سیستم با استفاده از الگوریتمEDF زمانبندی میشود. یک وظیفه با شرط اینکه absolute deadline آن از سقف سیستم کمتر باشد اجازه شروع اجرا را دارد.