Project 1: Non-preemptive CPU Scheduling

Due date: Midnight of Wednesday Feb 16, 2022

Project objective:

- Understand **non-preemptive** process scheduling algorithms by implementing them in Python.
- Simulate the CPU scheduling aspects of an Operating System kernel.
- Practice building simulations in Python and Jupyter Notebooks.

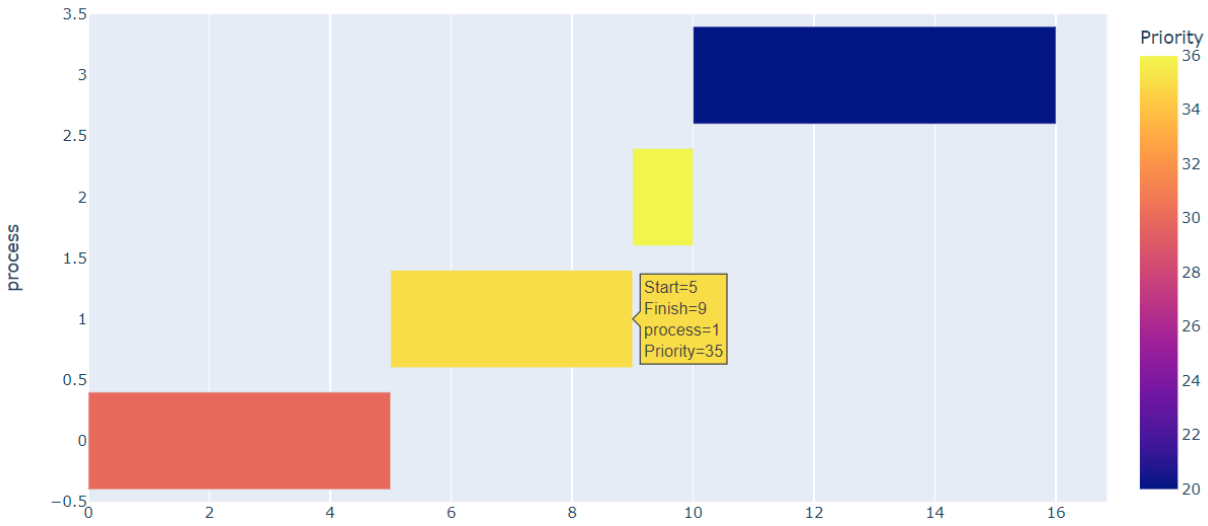

Figure 1: Gantt chart showing FCFS scheduling simulation of four processes over 16 time slices.

Project overview:

In this assignment, you will create simulations of the CPU scheduling aspects of an Operating System Kernel and implement the following scheduling algorithms:

1. FCFS: First-Come First-Served
2. SJF: Shortest Job First
3. Priority Scheduling

In addition to the algorithms themselves, you will write code to calculate statistics and simulations of specific scheduling situations that are associated with each algorithm. Finally, you will create a document in Jupyter Notebooks that acts as your **project report** where you run your tests, simulations, and create visualizations of each run. Figure 1 shows an example of a Gantt chart that is expected in your Notebook. This project is part of a three project series on CPU scheduling.

The Process Class:

We will need a structure to hold process information. Start by creating a new Python file and name it ***process.py***. In this file, you will implement the following:

1. A class called ***Process*** that implements a process in the ready state for scheduling.
2. The class has an ***__init__*** method that takes the parameters: id, burst_time, arrival_time, and priority.
3. The init method initializes attributes (fields) to hold process ID, process burst time, arrival time, and priority. In addition, it creates attributes to keep track of process **wait time** and **turnaround time**. These two attributes are initialized to zero.
4. The process class implements getters and setters for all fields except **id**, which only has a getter as ID does not change after process creation.

The scheduler file:

The ***scheduler.py*** file is where the magic happens, and you have the freedom to customize it and optimize it in any way you see fit as long as it works as intended in the end. The rest of this section will cover the way I suggest you go about writing your schedulers as functions in this file.

Starting with **FCFS**, write a function called *FCFS_scheduler* which takes the following parameters:

1. **processes**: is a list of all the processes in the simulation, whether they arrived or not, they are in this list.
2. **ready**: this is a list of processes with current arrival time. Meaning, if the arrival time of a process is less than the current time, it should be in the ready list. Therefore, this list holds only processes that have arrived at the ready list.
3. **CPU**: this is a list that simulates the CPU by holding beginning runtime and end of runtime for each process. This is the same as the Gantt bar that we have been using in lecture slides at the bottom of each example.
4. **time**: this is an integer that represents the current time, where simulation starts at time zero and time is incremented by one after each time slice.
5. **verbose**: this is a boolean with the default value of True. It controls a print statement that shows process ID, start time, and end time at each context switch. It is useful for debugging.

Below is the suggested FCFS scheduler function structure:

```
def FCFS_scheduler(processes, ready, CPU, time, verbose=True):
    ''' non-preemptive FCFS scheduler '''

    # pick process with lowest arrival time and remove it from ready

    # set start time to time

    # while process is not finished

        # decrement process burst time by one

        # add 1 to time

        # add processes that arrived now to ready queue

    # set end time to time

    # add processID, start, end to CPU (this will be useful later)
    CPU.append(dict(process=process.get_ID(),
                    Start=start_time,
                    Finish=end_time,
                    Priority=process.get_priority()))

    # return time
    return time
```

I suggest following the same general structure for all your schedulers. Also, I suggest writing helper functions such as:

1. **find_lowest_arrival**: this function takes one parameter, the ready list, and it returns the process with the lowest arrival time that should run next.
2. **add_ready**: this function takes the process list, the ready list, and current time, and it adds any processes that arrived to the ready list.

Note: All your schedulers go into this file, and you can write any helper functions that you want as long as you don't use any external code.

The operating_system file:

The ***operating_system.py*** file where processes are created, scheduler runs, and statistics are calculated. The structure of the file is rather simple, and it includes the following:

1. Import **process**, **scheduler**, and **pandas** as **pd**. Pandas is a data structure common in data science and research, it is easy to learn and we will use it to output our results into a file. Pandas will continue to appear in this course, you can read more about it here.

2. Create a function called kernel which simulates the CPU scheduling aspects of an operating system kernel. The function takes two parameters:

    a. selected_scheduler: which is one of your scheduler functions that you pass to the kernel to be used in the simulation.

    b. verbose: a boolean set to True by default that controls whether output messages should be printed from the scheduler or not.

3. Inside the kernel function, implement an empty list called ***CPU*** and an empty list called ***ready***.

4. Create an integer time variable and set it to zero.

5. Create a few processes for testing your schedulers, you can use the examples from class to verify your answers. Something like this should be a good starting point:

```
# creating processes
process0 = process.Process(0, 5, 0, 30)
process1 = process.Process(1, 4, 2, 35)
process2 = process.Process(2, 1, 5, 36)
process3 = process.Process(3, 6, 6, 20)
```

6. Create a list called ***processes*** and add the created processes to the list.

7. Add the ready processes to the ready list.

8. Run the scheduler until the ready list is empty.

9. Calculate wait-time and turnaround-time for each process, and calculate averages for the entire simulation.

10. Store the results in a pandas dataframe and then store the data frame as a CSV file. Here's some code that might be helpful:

```
    # save results as CSV
    df = pd.DataFrame(CPU)
    df.to_csv("results.csv", index=False)
```

    The ***results.csv*** file can be opened by Excel, and we will use it in the Jupyter Notebook to generate a Gantt chart visualization of each algorithm.

The Scheduling_Analyses Jupyter Notebook file:

The ***Scheduling_Analyses.ipynb*** file is where you run your tests, visualize the results of each scheduler, your simulations, and write your report. I will elaborate on each of the three parts separately.

Testing:

You are expected to import **operating_system.py** and **scheduler.py** into your notebook, and run the three schedulers and generate data to verify that the three schedulers work as intended. Let's use the four processes mentioned in the operating_system file section to verify the three algorithms.

To run a specific scheduler, all you have to do is run the kernel with the name of your scheduler like the example below:

operating_system.kernel(scheduler.FCFS_scheduler)

Since verbose is kept to the default true value, the line above should generate something like this:

process 0 starts: 0 ends: 5
process 1 starts: 5 ends: 9
process 2 starts: 9 ends: 10
process 3 starts: 10 ends: 16

If the CSV file was saved correctly, you should be able to open it in your Notebook using the following:

```
1 import pandas as pd
2
3 df = pd.read_csv("results.csv")
4
5 df.head()
```

	process	Start	Finish	Priority
0	0	0	5	30
1	1	5	9	35
2	2	9	10	36
3	3	10	16	20

Visualization:

The following code should generate the Gantt chart in Figure 1 (Feel free to use other visualization code):

```
import plotly.express as px
fig = px.timeline(df, x_start="Start", x_end="Finish", y="process",
color="Priority")

df['delta'] = df['Finish'] - df['Start']
fig.layout.xaxis.type = 'linear'
fig.data[0].x = df.delta.tolist()
fig.show()
```

Simulations:
Programmatically, create the following demonstrations:

1. Create a dataset consisting of 10 processes that demonstrates how FCFS is not fair to a certain type of processes. Your demonstration should clearly state the process type.
2. Create a dataset consisting of 10 processes that demonstrates how SJF is not fair to a certain type of processes. Your demonstration should clearly state the process type.
3. Create a dataset consisting of 10 processes that demonstrates **starvation** in Priority Scheduling.

Support your demonstrations with numbers and statistics, and make sure that your processes are generated programmatically (not one at a time like the example on page 4).

Note: You are allowed to make modifications to your code, including modifying your kernel function to take in a list of processes.

Report:
Your Jupyter Notebook should be a coherent document that includes markdown text and code to communicate the sections above in addition to:

1. Abstract: A brief summary of the project, in your own words. This should be no more than a few sentences. Give the reader context and identify the key purpose of the assignment.
2. Results: A section that goes over schedulers testing, visualization, and simulations.
3. Discussion: A section that interprets and describes the significance of your findings focussing on the simulation results.
4. Extensions: Describe any extensions you undertook, including text output, graphs, tables, or images demonstrating those extensions.
5. References/Acknowledgements.

Extensions:

You can be creative here and come up with your own extensions. I will suggest the following ideas:

- Implement ***aging*** with priority scheduling and show it with an example that would result in starvation without aging.
- Come up with your own scheduling algorithm and explain in which situations it might be useful/better.
- Create elaborate visualizations to show and compare the simulation results.
- Generate a large random dataset of processes and use it to compare the performance of the three schedulers.
- Add other metrics such as response time to your statistics.

Project submission:

- Add all your files (except *.ipynb_checkpoints* and *__pycache__*) to your ***project_1*** directory on Google Drive.
- Copy the project rubric from Moodle to the project directory after you review it.