과제 1

In [1]:
```
import json
import plotly.express as px
import plotly.figure_factory as ff # plotly를 사용하기 위해 pip install plotly
import pandas as pd                # plotly.express를 사용하려면 pandas도 필요함 pip install pandas
from plotly.figure_factory import create_gantt
import queue
```

In [2]:
```
inputfile = 'input1.json'     # 불러올 파일의 이름을 적어준다
with open(inputfile) as f:
    data = json.load(f)       # json파일을 불러와 data 변수에 저장해준다
data_len = len(data) # data의 길이를 data_len에 저장
```

비실시간 스케줄링과 실시간 스케줄링 구분

In [3]:
```
real_flag = 0
if 'period' in data[0].keys(): # period 라는 키 이름이 있으면 실시간 스케줄링으로 구분
    real_flag = 1
```

비실시간 스케줄링

1. First-Come, First-Served Scheduling(FCFS)

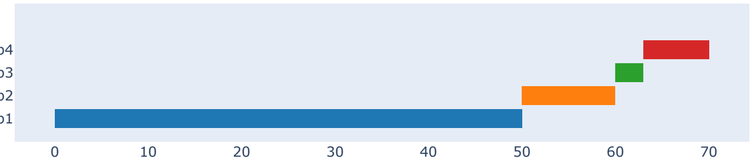
In [4]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
cpu_work = 0
cpu_work_flag = [0 for i in range(len(data))]
finish_flag = 0
dispatch = -1
#print(data[0]["burst_time"])

df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

for i in range(data_len):
    remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다
#print(remaining_time[1])

while True:
    for i in range(data_len):
        #print(i)
        if (data[i]["arrival_time"] <= time) and (cpu_work == 0) and (remaining_time[i] > 0): # CPU가 비어있을 때
            cpu_work_flag[i] = 1
            dispatch = dispatch + 1
        elif (data[i]["arrival_time"] <= time and (cpu_work == 1) and remaining_time[i] > 0 and cpu_work_flag[i] == 0): # CPU가 동작중일 때
            waiting_time[i] = waiting_time[i] + 1
            response_time[i] = response_time[i] + 1
            turnaround_time[i] = turnaround_time[i] + 1
            #print(waiting_time[i], i)

        if (cpu_work_flag[i] == 1): # cpu_work_flag가 1이면
            remaining_time[i] = remaining_time[i] - 1
            turnaround_time[i] = turnaround_time[i] + 1
            if remaining_time[i] >= 1: # remaining_time이 남았으면 cpu_work를 1로 만들어준다
                cpu_work = 1
            else:                      # remaining_time이 없으면 cpu_work_flag를 2로 만들어준다
                cpu_work_flag[i] = 2
```

```
        for j in range(data_len):
            if cpu_work_flag[j] == 2: # cpu_work_flag가 2인 프로세스가 있으면 (모두 실행된 프로세스가 있으면)
                cpu_work = 0
                cpu_work_flag[j] = 3  # 프로세스의 실행이 끝나면 cpu_work_flag는 3으로 만들어준다


        time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 cpu_work_flag가 3이면 finish_flag는 0이 된다
            if not cpu_work_flag[k] == 3:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break


    for i in range(data_len): # data의 길이만큼 for문 실행
        df.loc[i]=(data[i]["name"], response_time[i]+data[i]["arrival_time"], turnaround_time[i]+data[i]["arrival_time"]) # Task 열에는 가져온 데이터의 이름을 Start열에는 현재 time값을 Finish열에는 현재 time값에 Burst_time을 더한 값을 넣어준다
```

그래프 그리기

In [5]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

Gantt Chart



분석

In [6]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    FCFS_turnaround_time = 0
    FCFS_waiting_time = 0
    FCFS_response_time = 0

    for i in range(data_len):
        FCFS_turnaround_time = FCFS_turnaround_time + turnaround_time[i]
        FCFS_waiting_time = FCFS_waiting_time + waiting_time[i]
        FCFS_response_time = FCFS_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', FCFS_turnaround_time)
    print('전체대기 시간:', FCFS_waiting_time)
    print('전체응답 시간:', FCFS_response_time)
    print('\n')
    print('평균총처리 시간:', FCFS_turnaround_time/data_len)
    print('평균대기 시간:', FCFS_waiting_time/data_len)
    print('평균응답 시간:', FCFS_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 3
처리량: 0.05714285714285714


전체총처리 시간: 203
전체대기 시간: 133
전체응답 시간: 133


평균총처리 시간: 50.75
평균대기 시간: 33.25
평균응답 시간: 33.25
```

2. Shortest-Job-First Scheduling(SJF)

비선점 SJF

In [7]:

```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    cpu_work = 0
    cpu_work_flag = [0 for i in range(len(data))]
    finish_flag = 0
    dispatch = -1
    short = float('inf')
    short_flag = [0 for i in range(len(data))]

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

    for i in range(data_len):
        remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

while True:
    short = float('inf')
    #short_flag = [0 for i in range(len(data))]
    for l in range(data_len): # 도착한 프로세스 중 실행되지 않고 버스트타임이 가장 작은 프로세스를 찾아낸다
        if (data[l]["arrival_time"] <= time) and (data[l]["burst_time"] < short) and (cpu_work == 0) and (cpu_work_flag[l] == 0):
            short = data[l]["burst_time"]
            short_flag = [0 for i in range(len(data))] # short_flag 초기화
            short_flag[l] = 1

    for i in range(data_len):
        #print(i)
        if (data[i]["arrival_time"] <= time) and (cpu_work == 0) and (remaining_time[i] > 0) and (short_flag[i] == 1): # CPU가 비어있을 때
            cpu_work_flag[i] = 1
            dispatch = dispatch + 1
        elif (data[i]["arrival_time"] <= time and remaining_time[i] > 0 and cpu_work_flag[i] == 0): # CPU가 동작중일 때
            waiting_time[i] = waiting_time[i] + 1
            response_time[i] = response_time[i] + 1
            turnaround_time[i] = turnaround_time[i] + 1
            #print(waiting_time[i], i)

        if (cpu_work_flag[i] == 1): # cpu_work_flag가 1이면
            remaining_time[i] = remaining_time[i] - 1
            turnaround_time[i] = turnaround_time[i] + 1
            if remaining_time[i] >= 1: # remaining_time이 남았으면 cpu_work를 1로 만들어준다
                cpu_work = 1
            else:                      # remaining_time이 없으면 cpu_work_flag를 2로 만들어준다
                cpu_work_flag[i] = 2

    for j in range(data_len):
        if cpu_work_flag[j] == 2: # cpu_work_flag가 2인 프로세스가 있으면 (모두 실행된 프로세스가 있으면)
            cpu_work = 0
            cpu_work_flag[j] = 3  # 프로세스의 실행이 끝나면 cpu_work_flag는 3으로 만들어준다
```

```
        time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 cpu_work_flag가 3이면 finish_flag는 0이 된다
            if not cpu_work_flag[k] == 3:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break

    for i in range(data_len): # data의 길이만큼 for문 실행
        df.loc[i]=(data[i]["name"], response_time[i]+data[i]["arrival_time"], turnaround_time[i]+data[i]["arrival_time"]) # Task 열에는 가져온 데이터의 이름을 Start열에는 현재 time값을 Finish열에는 현재 time값에 Burst_time을 더한 값을 넣어준다
```

그래프 그리기

In [8]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```


분석

In [9]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    NPSJF_turnaround_time = 0
    NPSJF_waiting_time = 0
    NPSJF_response_time = 0

    for i in range(data_len):
        NPSJF_turnaround_time = NPSJF_turnaround_time + turnaround_time[i]
        NPSJF_waiting_time = NPSJF_waiting_time + waiting_time[i]
        NPSJF_response_time = NPSJF_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', NPSJF_turnaround_time)
    print('전체대기 시간:', NPSJF_waiting_time)
    print('전체응답 시간:', NPSJF_response_time)
    print('\n')
    print('평균총처리 시간:', NPSJF_turnaround_time/data_len)
    print('평균대기 시간:', NPSJF_waiting_time/data_len)
    print('평균응답 시간:', NPSJF_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 3
처리량: 0.05714285714285714


전체총처리 시간: 193
전체대기 시간: 123
전체응답 시간: 123


평균총처리 시간: 48.25
평균대기 시간: 30.75
평균응답 시간: 30.75
```

선점 SJF

In [10]:

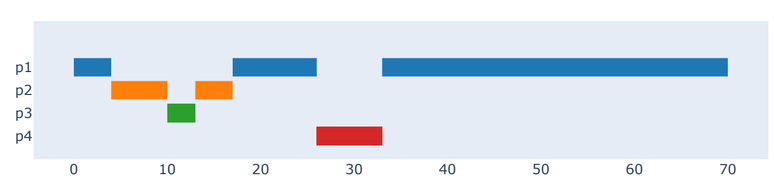
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    cpu_work = 0
    cpu_work_flag = [0 for i in range(len(data))]
    finish_flag = 0
    dispatch = -1
    short = float('inf')
    short_flag = [0 for i in range(len(data))]
    fig_name = []
    fig_start = []
    fig_stop = []

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

    for i in range(data_len):
        remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

    while True:
        short = float('inf')
        for l in range(data_len): # 도착한 프로세스 중 남은 버스트타임이 가장 작은 프로세스를 찾아낸다
            if (data[l]["arrival_time"] <= time) and (remaining_time[l] < short) and (remaining_time[l] > 0):
                short = remaining_time[l]
                short_flag = [0 for i in range(len(data))] # short_flag 초기화
                short_flag[l] = 1

        for i in range(data_len):
            #print(i)
            if (data[i]["arrival_time"] <= time) and (remaining_time[i] > 0) and (short_flag[i] == 1) and (cpu_work_flag[i] == 1): # 버스트타임이 가장 짧은 프로세스가 그대로 일때
                turnaround_time[i] = turnaround_time[i] + 1
                remaining_time[i] = remaining_time[i] - 1
                if remaining_time[i] == 0:
                    fig_stop.append(time + 1)
                    cpu_work_flag[i] = 0

            elif (data[i]["arrival_time"] <= time) and (remaining_time[i] > 0) and (short_flag[i] == 1) and (cpu_work_flag[i] == 0): #버스트타임이 가장 짧은 프로세스가 다른 프로세스 일 경우 (디스패치가 일어남)
                dispatch = dispatch + 1

                for j in range(data_len):
                    if cpu_work_flag[j] == 1:
                        cpu_work_flag[j] = 0
                        fig_stop.append(time)

                fig_name.append(data[i]["name"])
                fig_start.append(time)
                cpu_work_flag[i] = 1

                if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                    response_time[i] = time - data[i]["arrival_time"]
```

```
                turnaround_time[i] = turnaround_time[i] + 1
                remaining_time[i] = remaining_time[i] - 1

            elif (data[i]["arrival_time"] <= time and remaining_time[i] > 0): # 버스트타임이 가장 짧은 프로세스가 아닐 때
                waiting_time[i] = waiting_time[i] + 1
                turnaround_time[i] = turnaround_time[i] + 1
                #print(waiting_time[i], i)



        time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
            if remaining_time[k] > 0:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break

    for i in range(len(fig_name)): # data의 길이만큼 for문 실행
        df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [11]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

Gantt Chart



분석

In [12]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    PSJF_turnaround_time = 0
    PSJF_waiting_time = 0
    PSJF_response_time = 0

    for i in range(data_len):
        PSJF_turnaround_time = PSJF_turnaround_time + turnaround_time[i]
        PSJF_waiting_time = PSJF_waiting_time + waiting_time[i]
        PSJF_response_time = PSJF_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', PSJF_turnaround_time)
    print('전체대기 시간:', PSJF_waiting_time)
    print('전체응답 시간:', PSJF_response_time)
    print('\n')
    print('평균총처리 시간:', PSJF_turnaround_time/data_len)
    print('평균대기 시간:', PSJF_waiting_time/data_len)
    print('평균응답 시간:', PSJF_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 6
처리량: 0.05714285714285714


전체총처리 시간: 93
전체대기 시간: 23
전체응답 시간: 0


평균총처리 시간: 23.25
평균대기 시간: 5.75
평균응답 시간: 0.0
```

3. Round-Robin Scheduling

여기서 타임슬라이스는 4로 설정

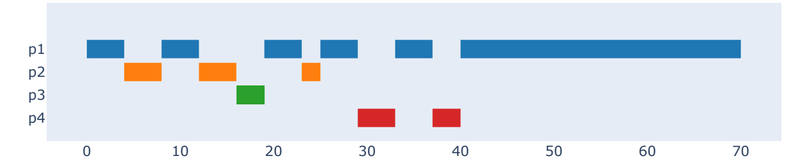
In [13]:

```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    finish_flag = 0
    dispatch = -1
    dispatch_flag = [0 for i in range(len(data))]
    fig_name = []
    fig_start = []
    fig_stop = []
    flag = 0

    time_slice = 4 # 타임슬라이스를 설정해준다
    run_time = [0 for i in range(len(data))]
    finish_time = [0 for i in range(len(data))]

    #print(data[0]["burst_time"])

df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

for i in range(data_len):
    remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다


while True:
    flag = 0
    for i in range(data_len):
        if (data[i]["arrival_time"] <= time and remaining_time[i] > 0): # 도착한 순서대로 프로세스를 실행
            flag = 1
            if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                dispatch = dispatch + 1

            if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                response_time[i] = time - data[i]["arrival_time"]

            remaining_time[i] = remaining_time[i] - time_slice

            fig_name.append(data[i]["name"])
            fig_start.append(time)
            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
                time = time + time_slice
                run_time[i] = run_time[i] + time_slice
                fig_stop.append(time)
```

```
                dispatch_flag = [0 for i in range(len(data))]
                dispatch_flag[i] = 1

        if flag == 0: # 아무 프로세스도 실행되지 않았으면
            time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
            if remaining_time[k] > 0:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break


    for i in range(len(fig_name)): # data의 길이만큼 for문 실행
        df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [14]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```



분석

In [15]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    RR_turnaround_time = 0
    RR_waiting_time = 0
    RR_response_time = 0

    for i in range(data_len):
        RR_turnaround_time = RR_turnaround_time + (finish_time[i] - data[i]["arrival_time"])
        RR_waiting_time = RR_waiting_time + (finish_time[i] - data[i]["arrival_time"] - run_time[i])
        RR_response_time = RR_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', RR_turnaround_time)
    print('전체대기 시간:', RR_waiting_time)
    print('전체응답 시간:', RR_response_time)
    print('\n')
    print('평균총처리 시간:', RR_turnaround_time/data_len)
    print('평균대기 시간:', RR_waiting_time/data_len)
    print('평균응답 시간:', RR_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 11
처리량: 0.05714285714285714


전체총처리 시간: 114
전체대기 시간: 44
전체응답 시간: 9


평균총처리 시간: 28.5
평균대기 시간: 11.0
평균응답 시간: 2.25
```

4. Priority Scheduling

비선점 Priority Scheduling

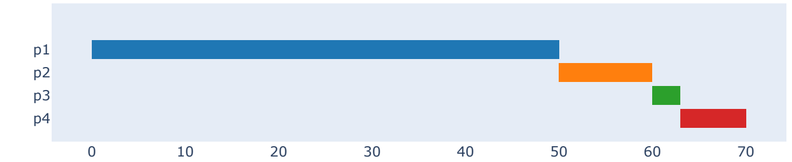
여기서는 낮은 수가 높은 우선순위를 나타낸다

In [16]:
```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    cpu_work = 0
    cpu_work_flag = [0 for i in range(len(data))]
    finish_flag = 0
    dispatch = -1
    short = float('inf')
    short_flag = [0 for i in range(len(data))]

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

    for i in range(data_len):
        remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

    while True:
        short = float('inf')
        for l in range(data_len): # 도착한 프로세스 중 실행되지 않고 우선순위가 가장 높은 프로세스를 실행시킨다
            if (data[l]["arrival_time"] <= time) and (data[l]["priority"] < short) and (cpu_work == 0) and (cpu_work_flag[l] == 0):
                short = data[l]["priority"]
                short_flag = [0 for i in range(len(data))] # short_flag 초기화
                short_flag[l] = 1

        for i in range(data_len):
            #print(i)
            if (data[i]["arrival_time"] <= time) and (cpu_work == 0) and (remaining_time[i] > 0) and (short_flag[i] == 1): # CPU가 비어있을 때
                cpu_work_flag[i] = 1
                dispatch = dispatch + 1
            elif (data[i]["arrival_time"] <= time and remaining_time[i] > 0 and cpu_work_flag[i] == 0): # CPU가 동작중일 때
                waiting_time[i] = waiting_time[i] + 1
                response_time[i] = response_time[i] + 1
                turnaround_time[i] = turnaround_time[i] + 1
                #print(waiting_time[i], i)

            if (cpu_work_flag[i] == 1): # cpu_work_flag가 1이면
                remaining_time[i] = remaining_time[i] - 1
                turnaround_time[i] = turnaround_time[i] + 1
                if remaining_time[i] >= 1: # remaining_time이 남았으면 cpu_work를 1로 만들어준다
                    cpu_work = 1
                else:                      # remaining_time이 없으면 cpu_work_flag를 2로 만들어준다
                    cpu_work_flag[i] = 2

        for j in range(data_len):
            if cpu_work_flag[j] == 2: # cpu_work_flag가 2인 프로세스가 있으면 (모두 실행된 프로세스가 있으면)
                cpu_work = 0
```

```
            cpu_work_flag[j] = 3  # 프로세스의 실행이 끝나면 cpu_work_flag는 3으로 만들어준다


        time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 cpu_work_flag가 3이면 finish_flag는 0이 된다
            if not cpu_work_flag[k] == 3:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break

    for i in range(data_len): # data의 길이만큼 for문 실행
        df.loc[i]=(data[i]["name"], response_time[i]+data[i]["arrival_time"], turnaround_time[i]+data[i]["arrival_time"]) # Task 열에는 가져온 데이터의 이름을 Start열에는 현재 time값을 Finish열에는 현재 time값에 Burst_time을 더한 값을 넣어준다
```

그래프 그리기

In [17]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```



분석

In [18]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    NPP_turnaround_time = 0
    NPP_waiting_time = 0
    NPP_response_time = 0

    for i in range(data_len):
        NPP_turnaround_time = NPP_turnaround_time + turnaround_time[i]
        NPP_waiting_time = NPP_waiting_time + waiting_time[i]
        NPP_response_time = NPP_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', NPP_turnaround_time)
    print('전체대기 시간:', NPP_waiting_time)
    print('전체응답 시간:', NPP_response_time)
    print('\n')
    print('평균총처리 시간:', NPP_turnaround_time/data_len)
    print('평균대기 시간:', NPP_waiting_time/data_len)
    print('평균응답 시간:', NPP_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 3
처리량: 0.05714285714285714


전체총처리 시간: 203
전체대기 시간: 133
전체응답 시간: 133


평균총처리 시간: 50.75
평균대기 시간: 33.25
평균응답 시간: 33.25
```

선점 Priority Scheduling

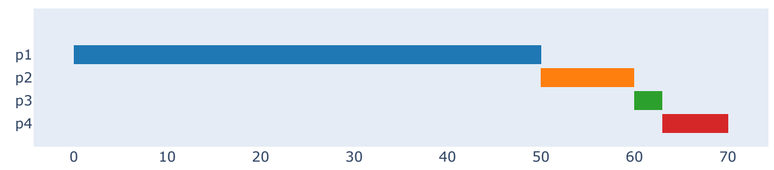
여기서는 낮은 수가 높은 우선순위를 나타낸다

In [19]:
```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    cpu_work = 0
    cpu_work_flag = [0 for i in range(len(data))]
    finish_flag = 0
    dispatch = -1
    short = float('inf')
    short_flag = [0 for i in range(len(data))]
    fig_name = []
    fig_start = []
    fig_stop = []

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

    for i in range(data_len):
        remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

    while True:
        short = float('inf')
        for l in range(data_len): # 도착한 프로세스 중 우선순위가 가장 높은 프로세스를 찾는다
            if (data[l]["arrival_time"] <= time) and (data[l]["priority"] < short) and (remaining_time[l] > 0):
                short = data[l]["priority"]
                short_flag = [0 for i in range(len(data))] # short_flag 초기화
                short_flag[l] = 1

        for i in range(data_len):
            #print(i)
            if (data[i]["arrival_time"] <= time) and (remaining_time[i] > 0) and (short_flag[i] == 1) and (cpu_work_flag[i] == 1): # 우선순위가 가장 높은 프로세스가 그대로 일때
                turnaround_time[i] = turnaround_time[i] + 1
                remaining_time[i] = remaining_time[i] - 1
                if remaining_time[i] == 0:
                    fig_stop.append(time + 1)
                    cpu_work_flag[i] = 0

            elif (data[i]["arrival_time"] <= time) and (remaining_time[i] > 0) and (short_flag[i] == 1) and (cpu_work_flag[i] == 0): #우선순위가 가장 높은 프로세스가 다른 프로세스 일 경우 (디스패치가 일어남)
                dispatch = dispatch + 1

                for j in range(data_len):
                    if cpu_work_flag[j] == 1:
                        cpu_work_flag[j] = 0
                        fig_stop.append(time)

                fig_name.append(data[i]["name"])
                fig_start.append(time)
                cpu_work_flag[i] = 1

                if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
```

```
                response_time[i] = time - data[i]["arrival_time"]

            turnaround_time[i] = turnaround_time[i] + 1
            remaining_time[i] = remaining_time[i] - 1

        elif (data[i]["arrival_time"] <= time and remaining_time[i] > 0): # 우선순위가 가장 높은 프로세스가 아닐 때
            waiting_time[i] = waiting_time[i] + 1
            turnaround_time[i] = turnaround_time[i] + 1
            #print(waiting_time[i], i)



    time = time + 1

    finish_flag = 0
    for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
        if remaining_time[k] > 0:
            finish_flag = 1

    if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
        break

for i in range(len(fig_name)): # data의 길이만큼 for문 실행
    df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [20]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```



분석

In [21]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    PP_turnaround_time = 0
    PP_waiting_time = 0
    PP_response_time = 0

    for i in range(data_len):
        PP_turnaround_time = PP_turnaround_time + turnaround_time[i]
        PP_waiting_time = PP_waiting_time + waiting_time[i]
        PP_response_time = PP_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', PP_turnaround_time)
    print('전체대기 시간:', PP_waiting_time)
    print('전체응답 시간:', PP_response_time)
    print('\n')
    print('평균총처리 시간:', PP_turnaround_time/data_len)
```

```
    print('평균대기 시간:', PP_waiting_time/data_len)
    print('평균응답 시간:', PP_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 3
처리량: 0.05714285714285714


전체총처리 시간: 203
전체대기 시간: 133
전체응답 시간: 133


평균총처리 시간: 50.75
평균대기 시간: 33.25
평균응답 시간: 33.25
```

5. Multilevel Queue Scheduling

여기서는 낮은 수가 높은 우선순위를 나타낸다
여기서 우선순위는 0 ~ 4까지 5가지로 나타낸다 (5개의 큐를 가진 다단계 큐 스케줄링 알고리즘이다)
각 큐는 Round-Robin으로 작동한다

In [22]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    finish_flag = 0
    dispatch = -1
    dispatch_flag = [0 for i in range(len(data))]
    count = 0
    time_slice = 4 # 타임슬라이스 설정
    break_flag = 0
    flag = 0

    run_time = [0 for i in range(len(data))]
finish_time = [0 for i in range(len(data))]

fig_name = []
fig_start = []
fig_stop = []

queue0 = queue.Queue() # 각 레벨 큐들을 생성
queue1 = queue.Queue()
queue2 = queue.Queue()
queue3 = queue.Queue()
queue4 = queue.Queue()

df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

for i in range(data_len):
    remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

while True:
    flag = 0
    queue0 = queue.Queue() # 큐 초기화
    queue1 = queue.Queue()
    queue2 = queue.Queue()
    queue3 = queue.Queue()
    queue4 = queue.Queue()

    for i in range(data_len): # 레디큐에 있는 프로세스들을 각 레벨에 맞는 큐에 넣어준다
        if (data[i]["arrival_time"] <= time) and (remaining_time[i] > 0):
            if data[i]["priority"] == 0:
                queue0.put(i)
            elif data[i]["priority"] == 1:
                queue1.put(i)
```

```
            elif data[i]["priority"] == 2:
                queue2.put(i)
            elif data[i]["priority"] == 3:
                queue3.put(i)
            elif data[i]["priority"] == 4:
                queue4.put(i)

        if queue0.qsize() > 0:
            flag = 1
            while queue0.qsize() > 0:
                i = queue0.get()
                if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                    dispatch = dispatch + 1

                if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                    response_time[i] = time - data[i]["arrival_time"]

                fig_name.append(data[i]["name"])
                fig_start.append(time)

                #remaining_time[i] = remaining_time[i] - time_slice
                for j in range(1, time_slice + 1):
                    remaining_time[i] = remaining_time[i] - 1
                    for k in range(data_len):
                        if (time + j) == data[k]["arrival_time"] and data[k]["priority"] == 0 and remaining_time[i] >= 0: # 우선순위가 같은 프로세스가 실행중 도착한다면 같은 큐에 추가시켜준다
                            queue0.put(k)

                if remaining_time[i] < 1: # 프로세스가 종료되면
                    time = time + (time_slice + remaining_time[i])
                    run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                    fig_stop.append(time)
                    finish_time[i] = time
                else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
                    time = time + time_slice
                    run_time[i] = run_time[i] + time_slice
                    fig_stop.append(time)

                dispatch_flag = [0 for i in range(len(data))]
                dispatch_flag[i] = 1
        elif queue1.qsize() > 0:
            flag = 1
            while queue1.qsize() > 0:
                i = queue1.get()
                if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                    dispatch = dispatch + 1

                if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                    response_time[i] = time - data[i]["arrival_time"]

                fig_name.append(data[i]["name"])
                fig_start.append(time)

                break_flag = 0
                for j in range(1, time_slice + 1):
                    remaining_time[i] = remaining_time[i] - 1
                    for k in range(data_len):
                        if (time + j) == data[k]["arrival_time"] and data[k]["priority"] == 1 and remaining_time[i] >= 0: # 우선순위가 같은 프로세스가 실행중 도착한다면 같은 큐에 추가시켜준다
                            queue1.put(k)
                        if (data[k]["arrival_time"] <= time + j) and (remaining_time[k] > 0) and (data[k]["priority"] < 1) and (remaining_time[i] >= 0): # 실행 도중 더 높은 우선순위를 가지는 프로세스가 도착하는 경우 CPU를 넘겨준다
                            time = time + j
                            run_time[i] = run_time[i] + j
                            fig_stop.append(time)
                            break_flag = 1
                            break
                    if break_flag == 1:
                        break
                if break_flag == 1: # while문을 탈출한다
```

```
                break
            #remaining_time[i] = remaining_time[i] - time_slice
            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
                time = time + time_slice
                run_time[i] = run_time[i] + time_slice
                fig_stop.append(time)

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1
    elif queue2.qsize() > 0:
        flag = 1
        while queue2.qsize() > 0:
            i = queue2.get()
            if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                dispatch = dispatch + 1

            if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                response_time[i] = time - data[i]["arrival_time"]

            fig_name.append(data[i]["name"])
            fig_start.append(time)

            break_flag = 0
            for j in range(1, time_slice + 1):
                remaining_time[i] = remaining_time[i] - 1
                for k in range(data_len):
                    if (time + j) == data[k]["arrival_time"] and data[k]["priority"] == 2 and remaining_time[i] >= 0: # 우선순위가 같은 프로세스가 실행중 도착한다면 같은 큐에 추가시켜준다
                        queue2.put(k)
                    if (data[k]["arrival_time"] <= time + j) and (remaining_time[k] > 0) and (data[k]["priority"]) < 2 and (remaining_time[i] >= 0): # 실행 도중 더 높은 우선순위를 가지는 프로세스가 도착하는 경우 CPU를 넘겨준다
                        time = time + j
                        run_time[i] = run_time[i] + j
                        fig_stop.append(time)
                        break_flag = 1
                        break
                if break_flag == 1:
                    break
            if break_flag == 1: # while문을 탈출한다
                break

            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
                time = time + time_slice
                run_time[i] = run_time[i] + time_slice
                fig_stop.append(time)

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1
    elif queue3.qsize() > 0:
        flag = 1
        while queue3.qsize() > 0:
            i = queue3.get()
            if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                dispatch = dispatch + 1

            if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                response_time[i] = time - data[i]["arrival_time"]

            fig_name.append(data[i]["name"])
```

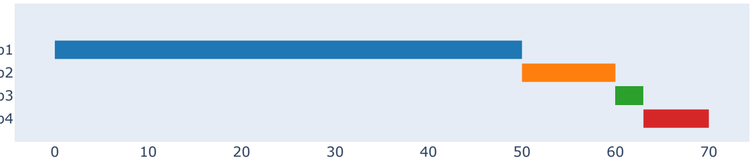
```
            fig_start.append(time)

            break_flag = 0
            for j in range(1, time_slice + 1):
                remaining_time[i] = remaining_time[i] - 1
                for k in range(data_len):
                    if (time + j) == data[k]["arrival_time"] and data[k]["priority"] == 3 and remaining_time[i] >= 0: # 우선순위가 같은 프로세스가 실행중 도착한다면 같은 큐에 추가시켜준다
                        queue3.put(k)
                    if (data[k]["arrival_time"] <= time + j) and (remaining_time[k] > 0) and (data[k]["priority"] < 3) and (remaining_time[i] >= 0): # 실행 도중 더 높은 우선순위를 가지는 프로세스가 도착하는 경우 CPU를 넘겨준다
                        time = time + j
                        run_time[i] = run_time[i] + j
                        fig_stop.append(time)
                        break_flag = 1
                        break
                if break_flag == 1:
                    break
            if break_flag == 1: # while문을 탈출한다
                break

            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
                time = time + time_slice
                run_time[i] = run_time[i] + time_slice
                fig_stop.append(time)

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1
    elif queue4.qsize() > 0:
        flag = 1
        while queue4.qsize() > 0:
            i = queue4.get()
            if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                dispatch = dispatch + 1

            if remaining_time[i] == data[i]["burst_time"]: # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다
                response_time[i] = time - data[i]["arrival_time"]

            fig_name.append(data[i]["name"])
            fig_start.append(time)

            break_flag = 0
            for j in range(1, time_slice + 1):
                remaining_time[i] = remaining_time[i] - 1
                for k in range(data_len):
                    if (time + j) == data[k]["arrival_time"] and data[k]["priority"] == 4 and remaining_time[i] >= 0: # 우선순위가 같은 프로세스가 실행중 도착한다면 같은 큐에 추가시켜준다
                        queue4.put(k)
                    if (data[k]["arrival_time"] <= time + j) and (remaining_time[k] > 0) and (data[k]["priority"] < 4) and (remaining_time[i] >= 0): # 실행 도중 더 높은 우선순위를 가지는 프로세스가 도착하는 경우 CPU를 넘겨준다
                        time = time + j
                        run_time[i] = run_time[i] + j
                        fig_stop.append(time)
                        break_flag = 1
                        break
                if break_flag == 1:
                    break
            if break_flag == 1: # while문을 탈출한다
                break

            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 레디큐로 돌아갈 경우
```

```
                time = time + time_slice
                run_time[i] = run_time[i] + time_slice
                fig_stop.append(time)

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1

    if flag == 0: # 아무 프로세스가 실행되지 않으면
        time = time + 1

    finish_flag = 0
    for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
        if remaining_time[k] > 0:
            finish_flag = 1

    if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
        break

for i in range(len(fig_name)): # data의 길이만큼 for문 실행
    df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [23]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

Gantt Chart



분석

In [24]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    MLQ_turnaround_time = 0
    MLQ_waiting_time = 0
    MLQ_response_time = 0

    for i in range(data_len):
        MLQ_turnaround_time = MLQ_turnaround_time + (finish_time[i] - data[i]["arrival_time"])
        MLQ_waiting_time = MLQ_waiting_time + (finish_time[i] - data[i]["arrival_time"] - run_time[i])
        MLQ_response_time = MLQ_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', MLQ_turnaround_time)
    print('전체대기 시간:', MLQ_waiting_time)
    print('전체응답 시간:', MLQ_response_time)
    print('\n')
    print('평균총처리 시간:', MLQ_turnaround_time/data_len)
    print('평균대기 시간:', MLQ_waiting_time/data_len)
    print('평균응답 시간:', MLQ_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 3
처리량: 0.05714285714285714


전체총처리 시간: 203
전체대기 시간: 133
전체응답 시간: 133


평균총처리 시간: 50.75
평균대기 시간: 33.25
평균응답 시간: 33.25
```

6. Multilevel Feedback Queue Scheduling

3개의 큐를 가지는 MLFQ
여기서 첫번째 큐의 타임슬라이스는 4이고 두번째 큐의 타임슬라이스는 8이다 마지막 큐는 FCFS로 작동한다

In [25]:

```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    time = 0
    waiting_time = [0 for i in range(len(data))]  # 대기 시간
    response_time = [0 for i in range(len(data))] # 응답 시간
    turnaround_time = [0 for i in range(len(data))] # 총처리 시간
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    finish_flag = 0
    dispatch = -1
    dispatch_flag = [0 for i in range(len(data))]
    count = 0
    time_slice0 = 4 # 타임슬라이스 설정
    time_slice1 = 8
    time_count = [0 for i in range(len(data))]
    break_flag = 0
    flag = 0

    run_time = [0 for i in range(len(data))]
    finish_time = [0 for i in range(len(data))]

    fig_name = []
    fig_start = []
    fig_stop = []

    queue0 = queue.Queue() # 각 레벨 큐들을 생성
    queue1 = queue.Queue()
    queue2 = queue.Queue()

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['arrival_time']) # data를 arrival_time을 기준으로 오름차순으로 정렬해준다

    for i in range(data_len):
        remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

    while True:
        flag = 0

        for i in range(data_len): # 한번도 실행이 안된 레디큐에 도착한 프로세스를 큐에 넣어준다
            if (data[i]["arrival_time"] <= time) and (remaining_time[i] == data[i]["burst_time"]):
                queue0.put(i)


        if queue0.qsize() > 0:
            flag = 1
            while queue0.qsize() > 0: # 첫번째 큐
                i = queue0.get()
                if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                    dispatch = dispatch + 1

                response_time[i] = time - data[i]["arrival_time"] # 프로세스가 도착한 다음 처음 실행되는 시간까지 걸린 시간을 응답시간으로 저장해준다 (이 스케줄링에서는 무조건 맨 처음 큐에서 응답시간이 생긴다)

                fig_name.append(data[i]["name"])
```

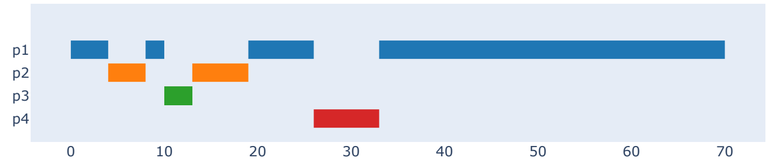
```
            fig_start.append(time)

            for j in range(1, time_slice0 + 1):
                remaining_time[i] = remaining_time[i] - 1
                for k in range(data_len):
                    if (time + j) == data[k]["arrival_time"] and remaining_time[i] >= 0: # 새로운 프로세스가 도착한다면 큐에 추가시켜 준다
                        queue0.put(k)

            if remaining_time[i] < 1: # 프로세스가 종료되면
                time = time + (time_slice0 + remaining_time[i])
                run_time[i] = run_time[i] + (time_slice0 + remaining_time[i])
                fig_stop.append(time)
                finish_time[i] = time
            else: # 프로세스가 종료되지 않고 다음 큐로 넘어갈 경우
                time = time + time_slice0
                run_time[i] = run_time[i] + time_slice0
                fig_stop.append(time)
                queue1.put(i)

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1

        elif queue1.qsize() > 0: # 두번째 큐
            flag = 1
            while queue1.qsize() > 0:
                i = queue1.get()
                if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                    dispatch = dispatch + 1

                fig_name.append(data[i]["name"])
                fig_start.append(time)

                break_flag = 0
                for j in range(1, time_slice1 + 1):
                    if (time_count[i] == time_slice1) and remaining_time[i] > 0: # 이미 시간할당량을 모두 쓴 프로세스는 다음 큐로 이동시켜준다
                        time = time + j - 1
                        run_time[i] = run_time[i] + j - 1
                        fig_stop.append(time)
                        queue2.put(i)
                        break_flag = 1
                        break
                    remaining_time[i] = remaining_time[i] - 1
                    time_count[i] = time_count[i] + 1
                    for k in range(data_len):
                        if (data[k]["arrival_time"] == time + j) and (remaining_time[k] > 0) and remaining_time[i] >= 0: # 실행 도중 새로운 프로세스가 도착한다면 CPU를 넘겨준다
                            time = time + j
                            run_time[i] = run_time[i] + j
                            fig_stop.append(time)
                            queue1.put(i)
                            break_flag = 1
                            break
                    if break_flag == 1:
                        break
                if break_flag == 1: # while문을 탈출한다
                    break
                #remaining_time[i] = remaining_time[i] - time_slice
                if remaining_time[i] < 1: # 프로세스가 종료되면
                    time = time + (time_slice1 + remaining_time[i])
                    run_time[i] = run_time[i] + (time_slice1 + remaining_time[i])
                    fig_stop.append(time)
                    finish_time[i] = time
                else: # 프로세스가 종료되지 않고 다음 큐로 넘어갈 경우
                    time = time + time_slice1
                    run_time[i] = run_time[i] + time_slice1
                    fig_stop.append(time)
                    queue2.put(i)
```

```
            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1


    elif queue2.qsize() > 0: # 세번째 큐
        flag = 1
        while queue2.qsize() > 0:
            i = queue2.get()
            if dispatch_flag[i] == 0: # 바로 전에 실행되었던 프로세스가 현재 프로세스와 다르다면 디스패치는 일어난다
                dispatch = dispatch + 1

            fig_name.append(data[i]["name"])
            fig_start.append(time)
           # print(i)
            break_flag = 0
            while remaining_time[i] > 0:
                remaining_time[i] = remaining_time[i] - 1
                time = time + 1
                run_time[i] = run_time[i] + 1
                for k in range(data_len):
                    if (data[k]["arrival_time"] == time) and (remaining_time[k] > 0) and (remaining_time[i] >= 0):  # 실행 도중 새로운 프로세스가 도착한다면 CPU를 넘겨준다
                        fig_stop.append(time)
                        queue2.put(i)
                        break_flag = 1
                        break
                if break_flag == 1:
                    break
            if break_flag == 1: # while문을 탈출한다
                break

            # 프로세스가 종료되면
            fig_stop.append(time)
            finish_time[i] = time

            dispatch_flag = [0 for i in range(len(data))]
            dispatch_flag[i] = 1


        if flag == 0: # 아무 프로세스가 실행되지 않으면
            time = time + 1

        finish_flag = 0
        for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
            if remaining_time[k] > 0:
                finish_flag = 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break

    for i in range(len(fig_name)): # data의 길이만큼 for문 실행
        df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [26]:
```
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

Gantt Chart



분석

In [27]:
```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    MLFQ_turnaround_time = 0
    MLFQ_waiting_time = 0
    MLFQ_response_time = 0

    for i in range(data_len):
        MLFQ_turnaround_time = MLFQ_turnaround_time + (finish_time[i] - data[i]["arrival_time"])
        MLFQ_waiting_time = MLFQ_waiting_time + (finish_time[i] - data[i]["arrival_time"] - run_time[i])
        MLFQ_response_time = MLFQ_response_time + response_time[i]

    print('디스패치 지연 발생 횟수:', dispatch) # 정확한 cpu 이용률을 알 수 없으므로 비슷하게 비교할 수 있는 디스패치 지연 발생 횟수로 비교
    print('처리량:', data_len/time)
    print('\n')
    print('전체총처리 시간:', MLFQ_turnaround_time)
    print('전체대기 시간:', MLFQ_waiting_time)
    print('전체응답 시간:', MLFQ_response_time)
    print('\n')
    print('평균총처리 시간:', MLFQ_turnaround_time/data_len)
    print('평균대기 시간:', MLFQ_waiting_time/data_len)
    print('평균응답 시간:', MLFQ_response_time/data_len)
```

```
디스패치 지연 발생 횟수: 8
처리량: 0.05714285714285714


전체총처리 시간: 95
전체대기 시간: 25
전체응답 시간: 0


평균총처리 시간: 23.75
평균대기 시간: 6.25
평균응답 시간: 0.0
```

결과 분석

In [28]:
```python
if real_flag == 0: # 비실시간 스케줄링 포맷이면
    print('선입 선처리 평균대기 시간:', FCFS_waiting_time/data_len)
    print('비선점 최단 작업 우선 평균대기 시간:', NPSJF_waiting_time/data_len)
    print('선점 최단 작업 우선 평균대기 시간:', PSJF_waiting_time/data_len)
    print('라운드 로빈 평균대기 시간:', RR_waiting_time/data_len)
    print('비선점 우선순위 평균대기 시간:', NPP_waiting_time/data_len)
    print('선점 우선순위 평균대기 시간:', PP_waiting_time/data_len)
    print('다단계 큐 평균대기 시간:', MLQ_waiting_time/data_len)
    print('다단계 피드백 큐 평균대기 시간:', MLFQ_waiting_time/data_len)

    waiting_time_list = [FCFS_waiting_time, NPSJF_waiting_time, PSJF_waiting_time, RR_waiting_time, NPP_waiting_time, PP_waiting_time, MLQ_waiting_time, MLFQ_waiting_time]
    waiting_time_name = ['선입 선처리', '비선점 최단 작업', '선점 최단 작업 우선', '라운드 로빈', '비선점 우선순위', '선점 우선순위', '다단계 큐', '다단계 피드백 큐']
    short = float('inf')
    for i in range(len(waiting_time_list)):
        if short > waiting_time_list[i]:
            short = waiting_time_list[i]
    for j in range(len(waiting_time_list)):
```

```
        if short == waiting_time_list[j]:
            print('\n최단 평균대기 시간을 갖는 스케줄링은: ', waiting_time_name[j], '입니다' )
```

```
선입 선처리 평균대기 시간: 33.25
비선점 최단 작업 우선 평균대기 시간: 30.75
선점 최단 작업 우선 평균대기 시간: 5.75
라운드 로빈 평균대기 시간: 11.0
비선점 우선순위 평균대기 시간: 33.25
선점 우선순위 평균대기 시간: 33.25
다단계 큐 평균대기 시간: 33.25
다단계 피드백 큐 평균대기 시간: 6.25

최단 평균대기 시간을 갖는 스케줄링은:  선점 최단 작업 우선 입니다
```

실시간 스케줄링

1. Rate-Monotonic Scheduling

여기서는 주기를 3번 반복한다 도착시간은 모두 동일하다
각 프로세스의 마감시간은 다음 주기가 시작하기 전까지이다

In [29]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    time = 0
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    finish_flag = 0
    fig_name = []
    fig_start = []
    fig_stop = []
    period_count = [1 for i in range(len(data))]
    error_flag = [0 for i in range(len(data))]
    break_flag = 0
    time_flag = 0
    short_flag = [0 for i in range(len(data))]
    start_flag = 0

    repeat = 3 # 주기를 3번 돈다

df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

data.sort(key=lambda x:x['period']) # data를 period 기준으로 오름차순으로 정렬해준다

for i in range(data_len):
    remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

while True:
    short = float('inf')
    short_flag = [0 for i in range(len(data))] # short_flag 초기화
    for l in range(data_len):
        if time == (data[l]["period"] * period_count[l]):
            if remaining_time[l] > 0:
                error_flag[l] = 1
                break_flag = 1
                break
            if period_count[l] <= repeat - 1:  # 아직 3번 반복하지 않았으면 준비상태로 바꾸어 준다
                remaining_time[l] = data[l]["burst_time"]
                period_count[l] = period_count[l] + 1
        if break_flag == 1:
            break
        if (data[l]["period"] < short) and (remaining_time[l] > 0): # period가 가장 짧은 준비상태의 프로세스
            short = data[l]["period"]
            short_flag = [0 for i in range(len(data))] # short_flag 초기화
            short_flag[l] = 1
    if break_flag == 1:
        break

    for i in range(data_len):
        if short_flag[i] == 1: # 가장 짧은 주기의 프로세스일 경우
            fig_name.append(data[i]["name"])
            fig_start.append(time)
```

```
                time = time + 1
                remaining_time[i] = remaining_time[i] - 1
                fig_stop.append(time)

        time_flag = 0
        for l in range(data_len): # 모든 프로세스가 쉬고있다면 시간 바꾸어 준다
            if short_flag[l] == 1:
                time_flag = 1

        finish_flag = 0
        for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
            if (period_count[k] <= repeat - 1) or (remaining_time[k] > 0):
                finish_flag = 1

        if time_flag == 0:
            time = time + 1

        if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
            break

    for i in range(len(fig_name)): # data의 길이만큼 for문 실행
        df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [30]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

분석

In [31]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    error = 0
    for i in range(data_len):
        if error_flag[i] == 1:
            error = 1
            print(data[i]["name"],'이(가) 마감시간 내에 처리되지 못했습니다.')
    if error == 0:
        print('모든 프로세스가 마감시간 내에 처리되었습니다')
```

2. Earliest-Deadline-First Scheduling

여기서는 주기를 3번 반복한다 도착시간은 모두 동일하다
각 프로세스의 마감시간은 다음 주기가 시작하기 전까지이다

In [32]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    time = 0
    remaining_time = [0 for i in range(len(data))] # 남은 시간
    finish_flag = 0
    fig_name = []
    fig_start = []
    fig_stop = []
    period_count = [1 for i in range(len(data))]
    error_flag = [0 for i in range(len(data))]
    break_flag = 0
    time_flag = 0
    short_flag = [0 for i in range(len(data))]
    start_flag = 0

    repeat = 3 # 주기를 3번 돈다

    df = pd.DataFrame(columns=['Task', 'Start', 'Finish']) # pd 데이터프레임의 열을 지정해준다

    data.sort(key=lambda x:x['period']) # data를 period 기준으로 오름차순으로 정렬해준다
```

```
for i in range(data_len):
    remaining_time[i] = data[i]["burst_time"] # remaining_time에 burst_time을 저장해준다

while True:
    short = float('inf')
    short_flag = [0 for i in range(len(data))] # short_flag 초기화
    for l in range(data_len):
        if time == (data[l]["period"] * period_count[l]):
            if remaining_time[l] > 0:
                error_flag[l] = 1
                break_flag = 1
                break
            if period_count[l] <= repeat - 1:  # 아직 3번 반복하지 않았으면 준비상태로 바꾸어 준다
                remaining_time[l] = data[l]["burst_time"]
                period_count[l] = period_count[l] + 1
        if break_flag == 1:
            break
        if (((data[l]["period"] * period_count[l])-time) < short) and (remaining_time[l] > 0): # 남은 마감시간이 가장 짧은 준비상태의 프로세스
            short = ((data[l]["period"] * period_count[l])-time)
            short_flag = [0 for i in range(len(data))] # short_flag 초기화
            short_flag[l] = 1
    if break_flag == 1:
        break

    for i in range(data_len):
        if short_flag[i] == 1: # 남은 마감시간이 가장 짧은 준비상태의 프로세스일 경우
            fig_name.append(data[i]["name"])
            fig_start.append(time)
            time = time + 1
            remaining_time[i] = remaining_time[i] - 1
            fig_stop.append(time)

    time_flag = 0
    for l in range(data_len): # 모든 프로세스가 쉬고있다면 시간 바꾸어 준다
        if short_flag[l] == 1:
            time_flag = 1

    finish_flag = 0
    for k in range(data_len): # 모든 프로세스가 작업이 끝났다면 finish_flag는 0이 된다
        if (period_count[k] <= repeat - 1) or (remaining_time[k] > 0):
            finish_flag = 1

    if time_flag == 0:
        time = time + 1

    if finish_flag == 0: # finish_flag가 0이면 루프를 중단시킨다
        break

for i in range(len(fig_name)): # data의 길이만큼 for문 실행
    df.loc[i]=(fig_name[i], fig_start[i], fig_stop[i])
```

그래프 그리기

In [33]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    fig = ff.create_gantt(df, index_col = 'Task', bar_width = 0.4, group_tasks=True) # df 데이터 프레임을 바탕으로 간트차트를 만들어준다
    fig.update_layout(xaxis_type='linear', autosize=False, width=800, height=300)
    fig.show()
```

분석

In [34]:
```
if real_flag == 1: # 실시간 스케줄링 포맷이면
    error = 0
    for i in range(data_len):
        if error_flag[i] == 1:
            error = 1
            print(data[i]["name"],'이(가) 마감시간 내에 처리되지 못했습니다.')
```

```
    if error == 0:
        print('모든 프로세스가 마감시간 내에 처리되었습니다')
```