CSC 314

OPERATING SYSTEM I

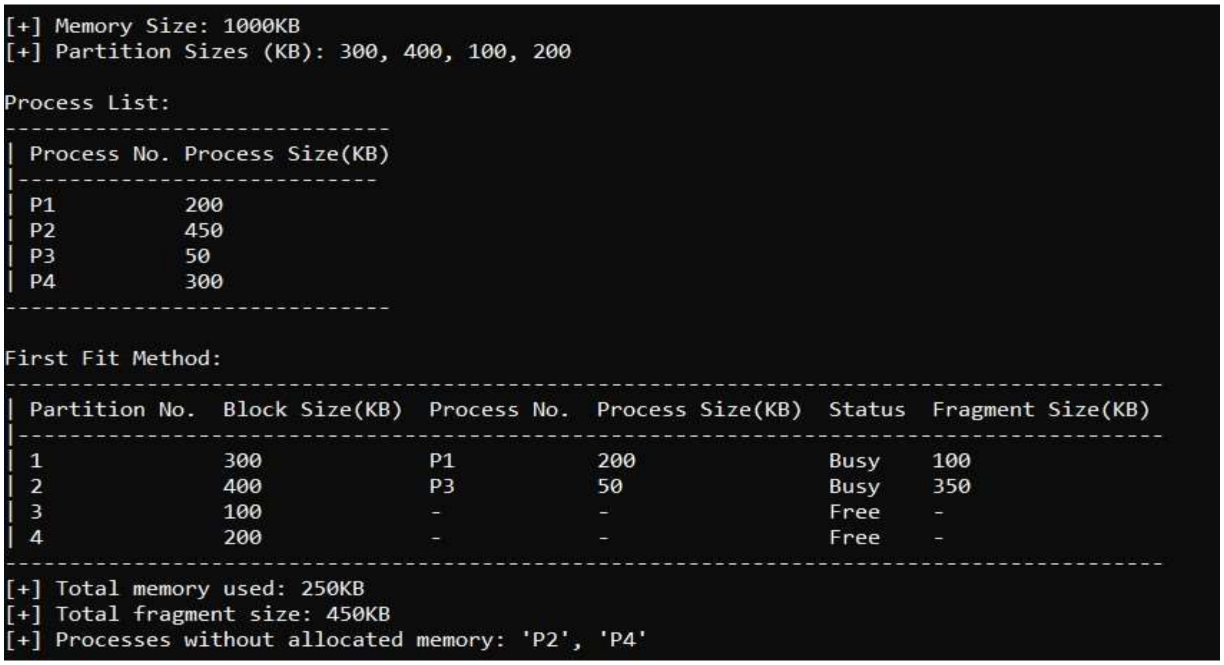
GROUP: 5

1.	MUMUNI ABDULLAH	180805047
2.	OLORUNFEMI-OJO TOMIWA	190805503
3.	ANIMASHAUN SOFIAT	190805520
4.	OGUNRINDE MOTUNRAYO	190805514
5.	ADEMUYIWA ABDULBAQI	190805522

TABLE OF CONTENTS

FIRST FIT MEMORY ALLOCATION SCHEME

```
#!/usr/bin/env python3

from __future__ import print_function

MEMORY_SIZE, NUMBER_OF_PARTITIONS = 1000, 4
PARTITION_SIZES = [300, 400, 100, 200]
PROCESS_SIZES = [200, 450, 50, 300]

def display_parameter_values():
    '''Displays the inputed parameters'''
    print(f"\n[+] Memory Size: {MEMORY_SIZE}KB")
    print(f"[+] Partition Sizes (KB): {str(PARTITION_SIZES)[1:-1]}\n")
    print(f"Process List:\n{'-'*30}")
    print("| {:<11} {:<16}".format("Process No.", "Process Size(KB)"))
    print(f"|{'-'*28}")
    for i, process_size in enumerate(PROCESS_SIZES):
        print("| {:<11} {:<16}".format(f"P{i + 1}", str(process_size)))
    print(f"{'-'*30}\n")

def display_output(output, total_used_memory, total_fragment_size, unallocated_processes):
    '''
    Tabulates the output
        args:
            output                 (dict): Key    (int) = Partition Number
                                           Value (list) = [Block Size, Process Number, Process 
Size, Process Status, Fragment Size]
            total_used_memory       (int): Sum of utilized memory
            total_fragment_size     (int): Sum of fragmented memory
            unallocated_processes (list): List of jobs without memory allocation
    '''
    print(f"First Fit Method:\n{'-'*90}")
    print(
        "| {:<14} {:<15} {:<12} {:<17} {:<7} {:<18}"
        .format("Partition No.", "Block Size(KB)", "Process No.", "Process Size(KB)", 
"Status", "Fragment Size(KB)")
    )
    print(f"|{'-'*89}")
    for key in sorted(output):
        print(
            "| {:<14} {:<15} {:<12} {:<17} {:<7} {:<18}"
            .format(key, str(output[key][0]), output[key][1], str(output[key][2]), 
output[key][3], str(output[key][4]))
        )
    print(f"{'-'*90}")
    print(f"[+] Total memory used: {total_used_memory}KB")
    print(f"[+] Total fragment size: {total_fragment_size}KB")
    print(f"[+] Processes without allocated memory: {str(unallocated_processes)[1:-1]}\n")

def first_fit():
    '''
    First fit algorithm implementation
        parameters:
            output                 (dict): Key    (int) = Partition Number
                                           Value (list) = [Block Size, Process Number, Process 
Size, Process Status, Fragment Size]
            is_partition_taken     (list): Tracks if a partition has been assigned a job
            is_process_x_taken     (dict): Tracks if a job has been assigned a memory location
            unallocated_processes (list): List of jobs without memory allocation
            total_used_memory      (int): Sum of utilized memory
            total_fragment_size    (int): Sum of fragmented memory
    '''
    output, is_partition_taken, unallocated_processes = {}, [False] * NUMBER_OF_PARTITIONS, []
    is_process_x_taken = {}
    total_used_memory, total_fragment_size = 0, 0
    for i, process_size in enumerate(PROCESS_SIZES):
        for j, partition_size in enumerate(PARTITION_SIZES):
            if is_partition_taken[j] == True:
                continue
            elif process_size <= partition_size:
                output[j + 1] = [partition_size, f"P{i + 1}", process_size, "Busy",
(partition_size - process_size)]
                total_used_memory += process_size
                total_fragment_size += (partition_size - process_size)
                is_partition_taken[j], is_process_x_taken[i + 1] = True, True
                break
            else:
                output[j + 1] = [partition_size, "-", "-", "Free", "-"]
                is_process_x_taken[i + 1] = False
    for key in is_process_x_taken:
        if is_process_x_taken[key] == False:
            unallocated_processes.append(f"P{key}")
    display_output(output, total_used_memory, total_fragment_size, unallocated_processes)

if __name__ == "__main__":
    display_parameter_values()
    first_fit()
```



```
[+] Memory Size: 1000KB
[+] Partition Sizes (KB): 300, 400, 100, 200

Process List:
------------------------------
| Process No. Process Size(KB)
|-----------------------------
| P1          200
| P2          450
| P3          50
| P4          300
------------------------------

First Fit Method:
------------------------------------------------------------------------------------------
| Partition No.  Block Size(KB)  Process No.  Process Size(KB)  Status  Fragment Size(KB)
|-----------------------------------------------------------------------------------------
| 1              300             P1           200               Busy    100
| 2              400             P3           50                Busy    350
| 3              100             -            -                 Free    -
| 4              200             -            -                 Free    -
------------------------------------------------------------------------------------------
[+] Total memory used: 250KB
[+] Total fragment size: 450KB
[+] Processes without allocated memory: 'P2', 'P4'
```

DOCUMENTATION

First Fit algorithm is used for allocating memory to processes. It allocates the first memory partition that is large enough to contain that process. For the implementation of this algorithm, some parameters are required:

1. `MEMORY_SIZE:` size of the block of memory to be partitioned and allocated to processes.
2. `NUMBER_OF_PARTTIONS:` number of partitions the memory is divided into.
3. `PARTITION_SIZES:` list of the sizes of the memory partitions.
4. `PROCESS_SIZES:` list of the sizes of the processes.

HOW DOES IT WORK?

The size of every process is compared to the sizes of all unclaimed partitions. The first partition that is greater than or equal to the size of the process is assigned.

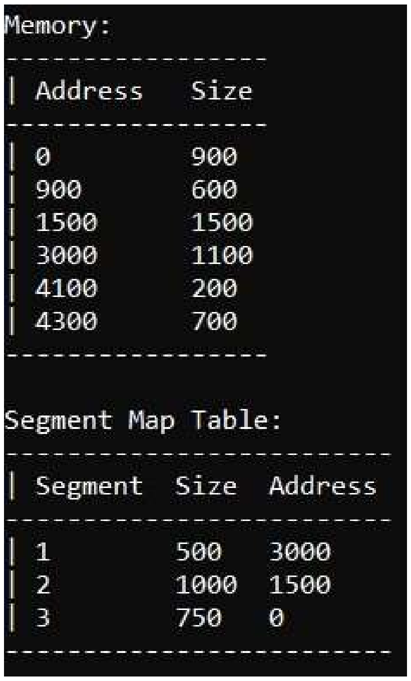
```
for i, process_size in enumerate(PROCESS_SIZES):
    for j, partition_size in enumerate(PARTITION_SIZES):
        if is_partition_taken[j] == True:
            continue  # checks if the partition has been claimed by another process
        elif process_size <= partition_size:
            # assign this partition to the process
            break
```

Considering the code output in page 5, a system has a memory block of size 1000KB. It is divided into 4 partitions of B1 (300KB), B2 (400KB), B3 (100KB) and B4 (200KB). The system has 4 jobs to perform, initialize a browsing session (P1 requires 200KB), start a video game (P2 - 450KB), play music (P3 - 50KB) and load a Word document (P4 - 300KB), The First Fit algorithm allocates P1 (200KB) to B1 (300KB) because it is the first partition large enough to initialize the browsing session. It wastes 100KB since it only requires 200KB. J2 i.e. the video game will not be started because there is no partition large enough among the free partitions to load the game. P3 is assigned to B2 (400KB). 350KB is wasted. B3 and B4 are not allocated any jobs. Out of the 1000KB, 250KB is used to perform 2 jobs, 450KB is wasted and 300KB is not allocated any job.

SEGMENTED MEMORY ALLOCATION SCHEME

```python
#!/usr/bin/env python3

from __future__ import print_function
import random

memory_size = 5000
memory = [
    {"address" : 0, "size" : 900},
    {"address" : 900, "size" : 600},
    {"address" : 1500, "size" : 1500},
    {"address" : 3000, "size" : 1100},
    {"address" : 4100, "size" : 200},
    {"address" : 4300, "size" : 700},
]
is_memory_claimed = [False] * 6
segments = [
    {"id" : 1, "size" : 500, "address" : None},
    {"id" : 2, "size" : 1000, "address" : None},
    {"id" : 3, "size" : 750, "address" : None}
]

def segmented_memory_allocation():
    for segment in segments:
        # Get a list of free memory locations
        available_locations = []
        for i, claimed in enumerate(is_memory_claimed):
            if not claimed: available_locations.append(i)
        # Get a list of partitions large enough to contain segment
        big_enough_locations = []
        for location in available_locations:
            if segment["size"] <= memory[location]["size"]:
                big_enough_locations.append(location)
        # Claim partition
        selected_partition = random.choice(big_enough_locations)
        is_memory_claimed[selected_partition] = True
        segment["address"] = memory[selected_partition]["address"]
    print("\nSegment Map Table: ")
    print("-"*25)
    print("| {:<8} {:<5} {:<7}".format("Segment", "Size", "Address"))
    print("-"*25)
    for segment in segments:
        print(
            "| {:<8} {:<5} {:<7}"
            .format(segment["id"], segment["size"], segment["address"])
        )
    print("-"*25)

if __name__ == "__main__":
    # Print memory addresses and their sizes
    print("Memory: ")
    print("-"*17)
    print("| {:<9} {:<5}".format("Address", "Size"))
    print("-"*17)
    for block in memory:
        print("| {:<9} {:<5}".format(block["address"], block["size"]))
    print("-"*17)
```

```
segmented_memory_allocation()
```



```
Memory:
-----------------
| Address   Size
-----------------
| 0         900
| 900       600
| 1500      1500
| 3000      1100
| 4100      200
| 4300      700
-----------------

Segment Map Table:
-------------------------
| Segment  Size  Address
-------------------------
| 1        500   3000
| 2        1000  1500
| 3        750   0
-------------------------
```

DOCUMENTATION

Segmented memory allocation is a memory management technique in which each process is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. A segment is a logical group of information such as a subroutine or data area. Each segment is actually a different logical address space of the program. Before segmented memory allocation scheme can be implemented, some details are needed. Namely:

- Beginning address of memory partitions and their sizes

```
memory = [
    {"address" : 0, "size" : 900},
    {"address" : 900, "size" : 600},
    {"address" : 1500, "size" : 1500},
    {"address" : 3000, "size" : 1100},
    {"address" : 4100, "size" : 200},
    {"address" : 4300, "size" : 700},
]
```

- Segment Map Table (comprises of segment number, size & address to be stored)

```
segments = [
    {"id" : 1, "size" : 500, "address" : None},
    {"id" : 2, "size" : 1000, "address" : None},
    {"id" : 3, "size" : 750, "address" : None}
]
```

HOW DOES IT WORK?

Before assigning a segment to a location, a list of the free memory locations is obtained so as not to overwrite data in a busy location.

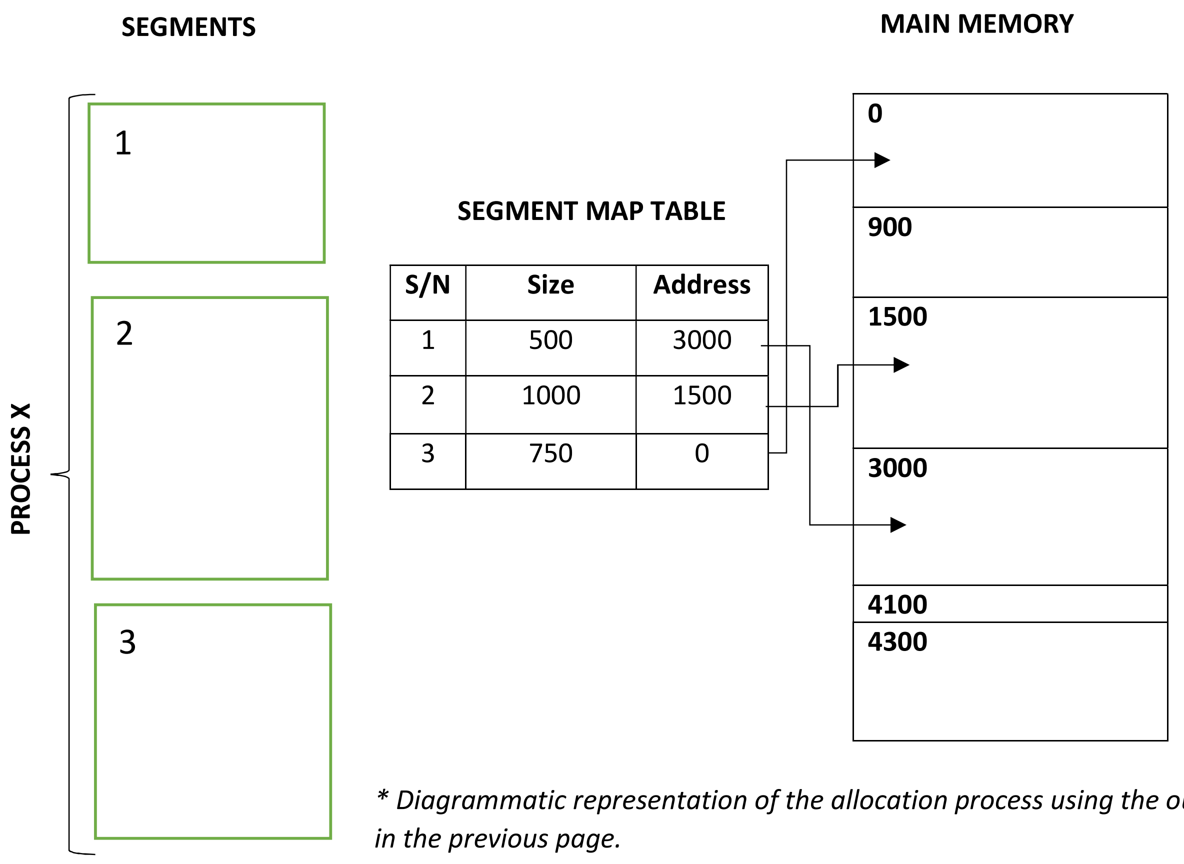
```
# Get a list of free memory locations
for segment in segments:
    available_locations = []
    for i, claimed in enumerate(is_memory_claimed):
        if not claimed: available_locations.append(i)
```

The next step is to compare the sizes of the free memory partitions to that of the segment to find partitions that are large enough for that segment.

```
# Get a list of partitions large enough to contain segment
big_enough_locations = []
for location in available_locations:
    if segment["size"] <= memory[location]["size"]:
        big_enough_locations.append(location)
```

After getting the list partitions large enough to contain the segment, one is randomly selected and the Segment Map table is updated to contain the location of the selected memory partition.

```
# Claim partition and update segment map table
selected_partition = random.choice(big_enough_locations)
is_memory_claimed[selected_partition] = True
segment["address"] = memory[selected_partition]["address"]
```

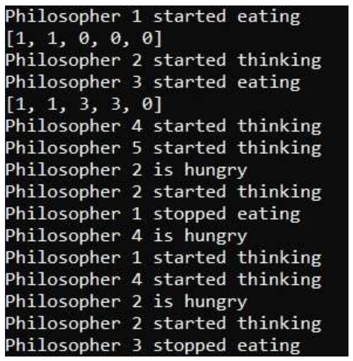


S/N	Size	Address
1	500	3000
2	1000	1500
3	750	0

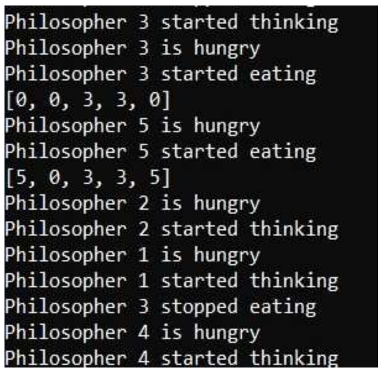
** Diagrammatic representation of the allocation process using the output in the previous page.*

DINING PHILOSOPHERS PROBLEM

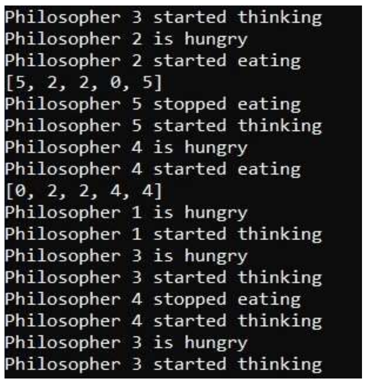
```python
#!/usr/bin/env python3

from __future__ import print_function
from threading import Thread, Lock
from time import sleep
import random

def eat(index, left_fork, right_fork):
    '''Pick up left & right fork and eat'''
    left_fork.acquire()
    right_fork.acquire()
    eating_philosophers[index % 5], eating_philosophers[(index + 1) % 5] = index + 1, index +
1
    print(f"Philosopher {index + 1} started eating")
    print(eating_philosophers)
    sleep(random.randint(1, 10))

def release_fork(index, left_fork, right_fork):
    '''Drop left & right fork'''
    left_fork.release()
    right_fork.release()
    eating_philosophers[index % 5], eating_philosophers[(index + 1) % 5] = 0, 0
    print(f"Philosopher {index + 1} stopped eating")

def think(index):
    '''Start thinking'''
    print(f"Philosopher {index + 1} started thinking")
    sleep(random.randint(1, 10))
    print(f"Philosopher {index + 1} is hungry")

def main(index, left_fork, right_fork):
    while True:
        # Eat if left & right fork are not claimed
        if not left_fork.locked() and not right_fork.locked():
            eat(index, left_fork, right_fork)
            release_fork(index, left_fork, right_fork)
        think(index)

if __name__ == "__main__":
    fork = [Lock() for x in range(5)]
    eating_philosophers = [0] * 5
    for i in range(5):
        Thread(target=main, args=(i, fork[i % 5], fork[(i + 1) % 5])).start()
```



```
Philosopher 1 started eating
[1, 1, 0, 0, 0]
Philosopher 2 started thinking
Philosopher 3 started eating
[1, 1, 3, 3, 0]
Philosopher 4 started thinking
Philosopher 5 started thinking
Philosopher 2 is hungry
Philosopher 2 started thinking
Philosopher 1 stopped eating
Philosopher 4 is hungry
Philosopher 1 started thinking
Philosopher 4 started thinking
Philosopher 2 is hungry
Philosopher 2 started thinking
Philosopher 3 stopped eating
```



```
Philosopher 3 started thinking
Philosopher 3 is hungry
Philosopher 3 started eating
[0, 0, 3, 3, 0]
Philosopher 5 is hungry
Philosopher 5 started eating
[5, 0, 3, 3, 5]
Philosopher 2 is hungry
Philosopher 2 started thinking
Philosopher 1 is hungry
Philosopher 1 started thinking
Philosopher 3 stopped eating
Philosopher 4 is hungry
Philosopher 4 started thinking
```



```
Philosopher 3 started thinking
Philosopher 2 is hungry
Philosopher 2 started eating
[5, 2, 2, 0, 5]
Philosopher 5 stopped eating
Philosopher 5 started thinking
Philosopher 4 is hungry
Philosopher 4 started eating
[0, 2, 2, 4, 4]
Philosopher 1 is hungry
Philosopher 1 started thinking
Philosopher 3 is hungry
Philosopher 3 started thinking
Philosopher 4 stopped eating
Philosopher 4 started thinking
Philosopher 3 is hungry
Philosopher 3 started thinking
```

DOCUMENTATION

Dining philosophers problem is a synchronization problem used to evaluate situations where there is need for allocating multiple resources to multiple resources. The problem states, there are five philosophers sitting around a circular table and their job is to either eat or think. The constraint is each philosopher is to eat with two forks but there are only five forks available. Each philosopher is to eat with the forks adjacent to him (fork to his left and right). There are not enough forks for every philosopher to eat at the same time. Considering the first image in the previous page, philosopher 1 starts eating. Philosopher 2 cannot eat because the fork to his left has been claimed. On line 8, philosopher 2 is hungry but cannot eat and hence is starving. As we can see from the other images, some philosophers will starve while some philosophers are eating.

HOW DOES IT WORK?

The basic implementation of each philosophers job is

```
while True:
    # Eat
    ...
    # Drop forks
    ...
    # Think
    ...
```

Each philosophers process of eating and thinking is independent, therefore, exists on its own thread.

```
# Create a thread for each philosopher and assign forks to be used
Thread(target=main, args=(philosopher_index, left_fork, right_fork])).start()
```

$$i = 0, \dots, 4$$

Their eating and thinking process are independent but they are sharing the same forks. Therefore, a Lock is used to represent each fork because it restricts access to only the thread that has claimed the fork.

```
# Creates five forks each represented as a Lock()
fork = [Lock() for x in range(5)]
```

Each philosopher is assigned fork $[i \,\%\, 5]$ and fork $[(i+1) \,\%\, 5]$. The modulus operation is used because the last philosopher will use the last and first fork to eat.

$$4 \,\%\, 5 = 4 \text{ and } (4+1)\% \, 5 = 0 \qquad i = 0, \dots, 4$$

```
fork = [Lock() for x in range(5)]
# Specify the fork each philosopher will use, then start their thread
for i in range(5):
    Thread(target=main, args=(i, fork[i % 5], fork[(i + 1) % 5])).start()
```

Each philosopher can only eat if the forks adjacent to him are free. If they are not, the philosopher begins to think. The is no time limit for eating or thinking

```
# Eat if left & right fork are not claimed
if not left_fork.locked() and not right_fork.locked():
    eat(index, left_fork, right_fork)
    release_fork(index, left_fork, right_fork)
think(index)
```

In order to eat, the philosopher picks up the forks adjacent to him.

```
left_fork.acquire()
right_fork.acquire()
print(f"Philosopher {index + 1} started eating")
```

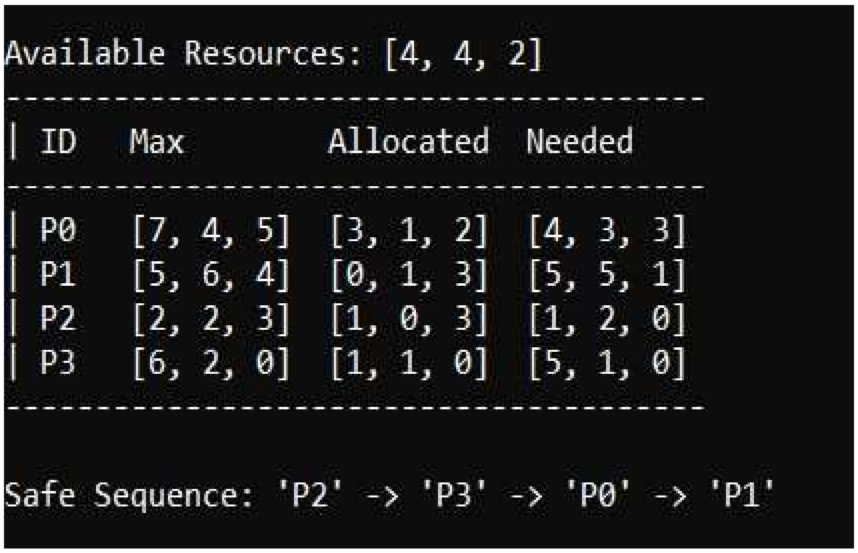
After eating, the philosopher drops the fork.

```
left_fork.release()
right_fork.release()
print(f"Philosopher {index + 1} stopped eating")
```

While some philosophers are eating, another philosopher that is ready to eat will be unable to eat because one or both forks will be in use. This philosopher is said to be Starving. He cannot get the resource that he needs. Since the choice of being ready and attempting to eat is solely up to philosopher (the choice of attempting to eat is not dependent on other philosophers) deadlock could arise. If all philosophers happen to decide to eat at the same time when all the forks are free, each will pick up the fork to the left and there will be deadlock because there will be no other forks available and none will release the already claimed one.

```
# Deadlock
eating_philosophers = [0, 1, 2, 3, 4]
```

```
#!/usr/bin/env python3

from __future__ import print_function

def bankers_algorithm():
    '''
    attr:
        deadlocked: (bool)
            Checks if deadlock has been ecountered
        safe_sequence: (list)
            Order to execute processes without encountering deadlock
        unexecuted_process: (list)
            List of unexecuted processes
    '''
    deadlocked, safe_sequence = False, []
    number_of_processes, number_of_resources  = 4, 3
    unexecuted_process = [x for x in range(number_of_processes)]
    available_resources = [4, 4, 2]
    max_resources = [
        [7, 4, 5],  # P0
        [5, 6, 4],  # P1
        [2, 2, 3],  # P2
        [6, 2, 0]   # P3
    ]
    allocated_resources = [
        [3, 1, 2],  # P0
        [0, 1, 3],  # P1
        [1, 0, 3],  # P2
        [1, 1, 0]   # P3
    ]
    resources_needed = []

    # Populate resources_needed list
    for i in range(number_of_processes):
        resources_needed_for_process_i = []
        for j in range(number_of_resources):
            resources_needed_for_process_i.append(max_resources[i][j] -
allocated_resources[i][j])
        resources_needed.append(resources_needed_for_process_i)

    print(f"Available Resources: {available_resources}\n")
    # Search for execution sequence void of deadlock
    while not deadlocked:
        initial_number_of_unexecuted_processes = len(unexecuted_process)
        for i in range(number_of_processes):
            # Check if process has been executed
            if i in safe_sequence:
                continue
            # Check if available resources are enough for process
            is_available_resource_enough = [False] * number_of_resources
            for j in range(number_of_resources):
                is_available_resource_enough[j] = available_resources[j] >=
resources_needed[i][j]
            if all(is_available_resource_enough):
                safe_sequence.append(i)
                unexecuted_process.remove(i)
                for j in range(number_of_resources):
                    available_resources[j] += allocated_resources[i][j]
        # Stop algorithm if all processes have been executed
        if len(unexecuted_process) == 0:
            break
        deadlocked = initial_number_of_unexecuted_processes == len(unexecuted_process)

    print("-"*39)
    print("| {:<4} {:<10} {:<10} {:<10}".format("ID", "Max", "Allocated", "Needed"))
    print("-"*39)
    for i in range(4):
        print(
            "| {:<4} {:<10} {:<10} {:<10}"
            .format(f"P{i}", str(max_resources[i]), str(allocated_resources[i]),
str(resources_needed[i]))
        )
    print("-"*39)

    if not deadlocked:
        print("\nSafe Sequence: ", end="")
        print(str([f"P{x}" for x in safe_sequence])[1:-1].replace(",", " ->"))
    elif deadlocked and len(safe_sequence) != 0:
        print("{} -> Deadlock".format(str([f"P{x}" for x in safe_sequence])[1:-1].replace(",",
" ->")))
    else:
        print("Deadlock!!!")

if __name__ == "__main__":
    bankers_algorithm()
```



```
Available Resources: [4, 4, 2]
---------------------------------------
| ID   Max        Allocated  Needed
---------------------------------------
| P0   [7, 4, 5]  [3, 1, 2]  [4, 3, 3]
| P1   [5, 6, 4]  [0, 1, 3]  [5, 5, 1]
| P2   [2, 2, 3]  [1, 0, 3]  [1, 2, 0]
| P3   [6, 2, 0]  [1, 1, 0]  [5, 1, 0]
---------------------------------------

Safe Sequence: 'P2' -> 'P3' -> 'P0' -> 'P1'
```

DOCUMENTATION

Bankers algorithm is a deadlock avoidance algorithm. It simulates resource allocation to processes to determine the safe sequence for allocating resources that will be void of deadlock (other processes enter a waiting state because a process has been allocated the resources but it is not enough and therefore cannot execute). Its original use was in the banking system to check if a loan should be given to a person or not. Banks use it to find a way to allocate money to its customers that wouldn't cause them to be unable to satisfy the needs of other customers. The algorithm compares the resources needed by a process for execution to the resources available. If it is enough, it gives the go ahead for the available resources to be allocated to the process. After the process finishes executing, all the resources allocated to that process are deallocated to become the new available resources and the algorithm repeats for the other processes.

HOW DOES IT WORK?

The amount of resources needed by each process is calculated by subtracting the maximum resources required from the amount of resources currently allocated to that process.

```
for i in range(number_of_processes):
    resources_needed_for_process_i = []
    for j in range(number_of_resources):
        resources_needed_for_process_i.append(max_resources[i][j] - allocated_resources[i][j])
    resources_needed.append(resources_needed_for_process_i)
```

Two lists are created, one to keep track of the unexecuted processes and the other to keep track of the safe sequence of execution.

```
unexecuted_process = [x for x in range(number_of_processes)]
safe_sequence = []
```

Bankers algorithm compares the resources needed by a process to what is available. If it is enough, the available resources are allocated to that process so that it can execute. Upon finishing execution, the resources are deallocated from the process and the new available resource becomes the sum of the initial available resources and initial resources assigned to the just executed process.

```
# Check if available resources are enough for process
is_available_resource_enough = [False] * number_of_resources
for j in range(number_of_resources):
    is_available_resource_enough[j] = available_resources[j] >= resources_needed[i][j]
if all(is_available_resource_enough):
    # Add to safe sequence
    safe_sequence.append(i)
    # Remove process from list of unexecuted processes
    unexecuted_process.remove(i)
    # Update the amount of available resources
    for j in range(number_of_resources):
        available_resources[j] += allocated_resources[i][j]
```

The algorithm compares the available resources to what is needed by unexecuted processes until one of the following occurs:

- All jobs are executed

```
                    while not deadlocked:
                        ...
                        ...
                        if len(unexecuted_process) == 0:
                            break
                        ...
```

- The available resources are not enough to for any of the unexecuted processes (Deadlock)

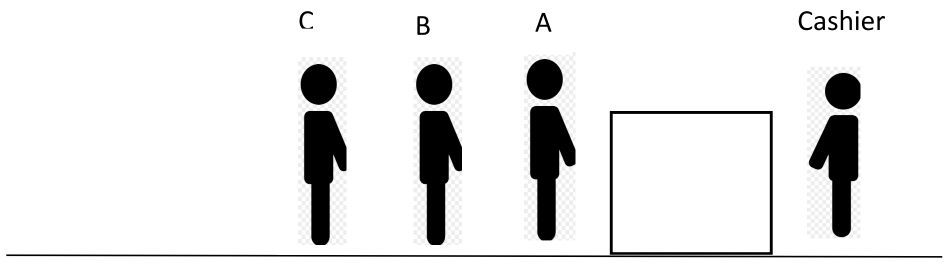
```
while not deadlocked:
        initial_number_of_unexecuted_processes = len(unexecuted_process)
            ...
            ...
        deadlocked = initial_number_of_unexecuted_processes == len(unexecuted_process)
```

Bankers algorithm helps avoid deadlock but has a few disadvantages:

- **High overhead:** the higher the number of processes, the higher the cost of checking for deadlock. It will take a longer time to run the algorithm and hence, it becomes inefficient.
- The algorithm requires a fixed number of processes.
- Processes must know the maximum amount of resources needed in advance.
- All of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. Starvation can arise.

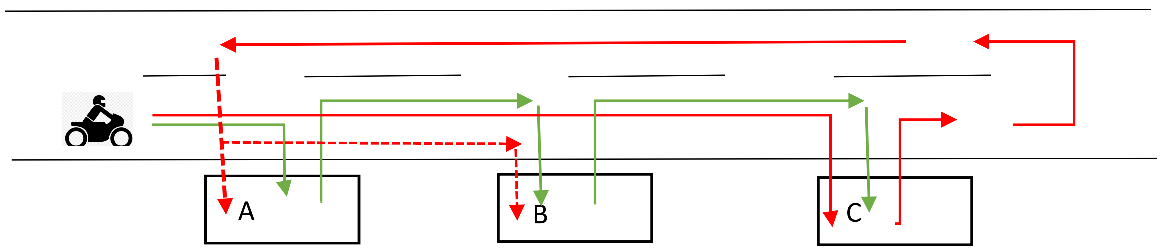
PROCESS SCHEDULING ALGORITHMS

1. **First-Come First-Serve:** A real-world example of FCFS is a single-lane one-way road, where the vehicle enters first, exits first or when buying a movie ticket, the first person in the queue is the first to be attended to.



**Cashier attending to customers. A is attended to first, followed by B then C.*

2. **Shortest Job First:** a real life example of SJF can be found in public transportation. The destination of a public transportation vehicle will always be the nearest bustop along a predefined route. The vehicle would not skip bustop A and B to drop passengers at bustop C on a straight road because going to bustop C is not the shortest job. Online delivery also uses this logic. The nearest order is delivered first, then after delivering the first order, it searches for the next nearest delivery location.



**Green arrow shows Shortest Job Next. The red path goes from C to A then to B which takes more time. SJF is therefore effective for delivery systems.*

3. **Priority Scheduling:** a real life example of Priority scheduling is the Order of Mathematical operations. Some mathematical operations have higher precedence over others and therefore, are solved first. The order of operation is Brackets, Orders, Division, Multiplication, Addition and Subtraction (BODMAS). Consider the problem below,

$$x = 2 \times 4 + 8 \div (9 - 5)$$

The Bracket is evaluated first because it is the operation with the highest priority.

$$x = 2 \times 4 + 8 \div 4$$

The next operation with the highest precedence is Division.

$$x = 2 \times 4 + 2$$

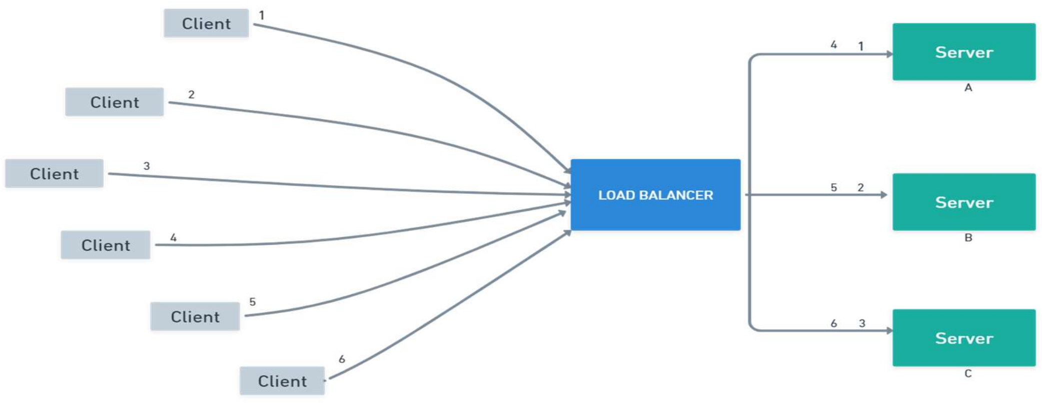
Multiplication is next in line.

$$x = 8 + 2$$

And finally, Addition

$$x = 10$$

4. **Shortest Remaining Time:** SRT can be observed at an accident scene. For example, two vehicles collide and several people are injured. Death is most likely to catch up with the most severely injured person because such an individual has the shortest time remaining to live. The more the severity of the injury, the shorter the time remaining to live. In this analogy, death represents the processor executing a process.
5. **Round Robin:** is used for distributing requests across a group of servers i.e. Load Balancing. Round robin is used to balance the server load and provide simple fault tolerance. Multiple identical servers are set up to deliver the same services or applications. When servers receive session requests, the requests are assigned using some criteria or in a rotating or random sequence to the available servers. An example of its implementation is the Least Connection load balancing technique. It assigns the latest request to the server that is currently servicing the fewest number of active sessions at that time. For example, there are two identical servers providing access to a service. User1 connects and initiates a session with server1. When User2 tries to initiate a session, the workload on both servers are checked and the one with least work is selected i.e. server2 at this particular point in time. Round robin's sequential movement of client requests across servers helps to manage high traffic.

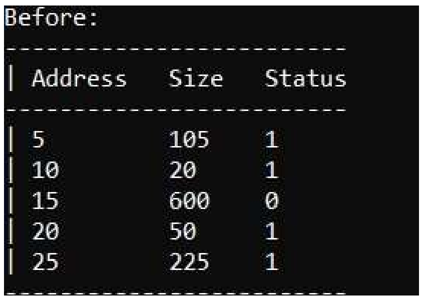


6. **Multiple Level Queue:** Real life example of Multi level Queue usage is in a manufacturing company. Let's say we have a machine that can be used to produce many different types of product. The machine must be setup when changing over from one product type to another. We want to minimize the time lost to setup (minimize the changeovers from one product to another) but still deliver orders on time. A multi-level queue approach is to group orders for the same product. We cannot keep adding orders to the first queue, as then orders in the other queues will be late, so at some point we will start adding orders of that type to a new queue with lower priority. We may also have some high priority queues for "rush" orders.

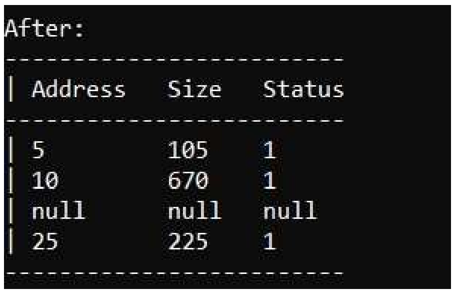
DEALLOCATION

(block to be deallocated is between 2 free blocks)

```python
#!/usr/bin/env python3

from __future__ import print_function

memory = [
    {"addr" : 5, "size" : 105, "free" : True},
    {"addr" : 10, "size" : 20, "free" : True},
    {"addr" : 15, "size" : 600, "free" : False},
    {"addr" : 20, "size" : 50, "free" : True},
    {"addr" : 25, "size" : 225, "free" : True}
]

def deallocation():
    '''
    Implementaion of deallocation scheme when the
    block to be deallocated is b/w two free blocks.
    '''
    display_output("Before: ", memory)

    for i, block in enumerate(memory):
        # Check if the block is busy and not the last block
        if not block["free"] and (i != len(memory) - 1):
            # Check if adjacent blocks are free
            if memory[i - 1]["free"] and memory[i + 1]["free"]:
                # Add block and next block size to previous block
                memory[i - 1]["size"] += memory[i]["size"] + memory[i + 1]["size"]
                # Set next block to null
                memory[i + 1]["addr"] = "null"
                memory[i + 1]["size"], memory[i + 1]["free"] = "null", "null"
                # Delete block
                del memory[i]

    display_output("After: ", memory)

def display_output(period, memory):
    '''Displays memory info'''
    print(f"{period}\n{'-'*25}")
    print("| {:<9} {:<6} {:<6}".format("Address", "Size", "Status"))
    print("-"*25)
    for block in memory:
        print("| {:<9} {:<6} {:<6}".format(block["addr"], block["size"], block["free"]))
    print("-"*25)

if __name__ == "__main__":
    deallocation()
```



```
Before:
-------------------------
| Address   Size   Status
-------------------------
| 5         105    1
| 10        20     1
| 15        600    0
| 20        50     1
| 25        225    1
-------------------------
```



```
After:
-------------------------
| Address   Size   Status
-------------------------
| 5         105    1
| 10        670    1
| null      null   null
| 25        225    1
-------------------------
```

DOCUMENTATION

Deallocation of memory is the act of freeing a block of memory used by a completed process. The finished process is removed from memory so that the partition can be utilized for something else. There are three possible scenarios encountered when deallocating a block from memory:

- Partition to be deallocated is adjacent to a free block
- Partition to be deallocated is between two free blocks
- Partition to be deallocated is isolated from other free blocks

In the second scenario, the partition above and below the partition to be deallocated are free.

```
memory = [
    ...
    {"addr" : 10, "size" : 20, "free" : True},
    # Partition to be deallocated
    {"addr" : 15, "size" : 600, "free" : False},
    {"addr" : 20, "size" : 50, "free" : True},
    ...
]
```

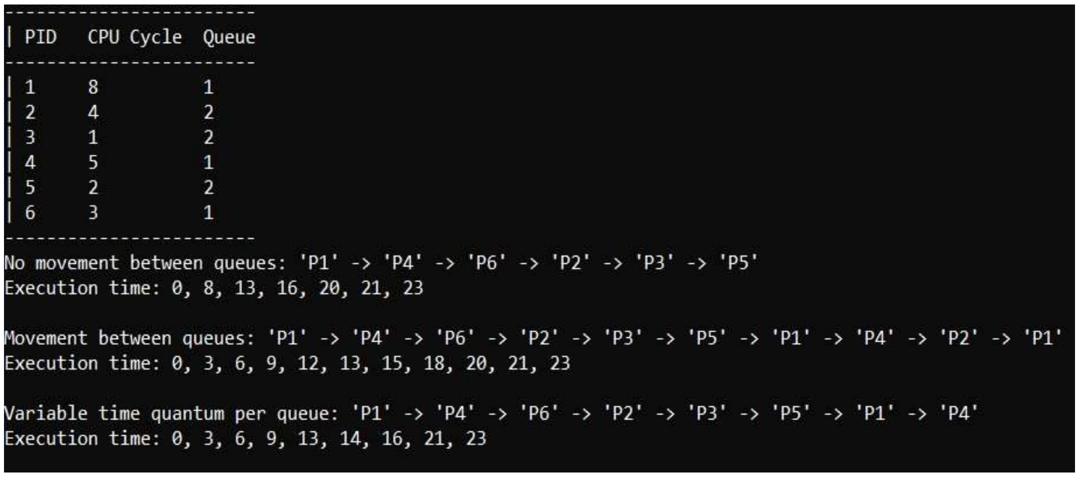
HOW DOES IT WORK?

Consider three blocks of memory A, B and C. B is the block to be deallocated, A is a free block above B and C is a free block below B. The sizes of these partitions combine to form a block and the address is set to that of the partition with the lowest beginning address, which is A. C is emptied out i.e. its address, size and status are set to NULL.

```
# Check if adjacent blocks are free
if memory[i - 1]["free"] and memory[i + 1]["free"]:
    # Add block and next block size to previous block
    memory[i - 1]["size"] += memory[i]["size"] + memory[i + 1]["size"]
    # Set next block to null
    memory[i + 1]["addr"] = "null"
    memory[i + 1]["size"], memory[i + 1]["free"] = "null", "null"
    # Delete block
    del memory[i]
```

MULTIPLE LEVEL QUEUES

```
#!/usr/bin/env python3

from __future__ import print_function
import copy

jobs = [
    {"id" : 1, "cpu cycle" : 8, "queue" : 1},
    {"id" : 2, "cpu cycle" : 4, "queue" : 2},
    {"id" : 3, "cpu cycle" : 1, "queue" : 2},
    {"id" : 4, "cpu cycle" : 5, "queue" : 1},
    {"id" : 5, "cpu cycle" : 2, "queue" : 2},
    {"id" : 6, "cpu cycle" : 3, "queue" : 1}
]

# NB: Queue 1 has higher priority than Queue 2
queue1, queue2 = [], []

def no_movement_btw_queues(queue1, queue2):
    '''Multiple level scheduling with no movement between queues'''
    priority, execution_sequence = [queue1, queue2], []
    execution_time = [0]
    for queue in priority:
        for job in queue:
            execution_sequence.append(job["id"])
            execution_time.append(execution_time[-1] + job["cpu cycle"])
    print("No movement between queues: ", end="")
    print(str([f"P{x}" for x in execution_sequence])[1:-1].replace(",", " ->"))
    print(f"Execution time: {str(execution_time)[1:-1]}\n")


def movement_btw_queues(queue1, queue2):
    '''Multiple level scheduling with movement between queues'''
    time_slice = 3
    priority, execution_sequence = [queue1, queue2], []
    execution_time = [0]
    for queue in priority:
        for job in queue:
            execution_sequence.append(job["id"])
            # Keep track of execution timeline
            if job["cpu cycle"] - time_slice > 0:
                execution_time.append(execution_time[-1] + time_slice)
            else:
                execution_time.append(execution_time[-1] + job["cpu cycle"])
            # Check if job has finished executing
            job["cpu cycle"] -= time_slice
            if job["cpu cycle"] > 0:
                # Add job to next queue
                if i == len(priority) - 1:
                    priority[-1].append(job)
                else:
                    priority[i + 1].append(job)
            else:
                del job
    print("Movement between queues: ", end="")
    print(str([f"P{x}" for x in execution_sequence])[1:-1].replace(",", " ->"))
    print(f"Execution time: {str(execution_time)[1:-1]}\n")

def variable_time_quantum_per_queue(queue1, queue2):
    '''Multiple level scheduling with variable time quantum per queue'''
    time_slice = 3
    priority, execution_sequence = [queue1, queue2], []
    execution_time = [0]
    for queue in priority:
        for job in queue:
            execution_sequence.append(job["id"])
            # Keep track of execution timeline
            if job["cpu cycle"] - time_slice > 0:
                execution_time.append(execution_time[-1] + time_slice)
            else:
                execution_time.append(execution_time[-1] + job["cpu cycle"])
            # Check if job has finished executing
            job["cpu cycle"] -= time_slice
            if job["cpu cycle"] > 0:
                # Add job to next queue
                if i == len(priority) - 1:
                    priority[-1].append(job)
                else:
                    priority[i + 1].append(job)
            else:
                del job
        # Double time slice
        time_slice += time_slice
    print("Variable time quantum per queue: ", end="")
    print(str([f"P{x}" for x in execution_sequence])[1:-1].replace(",", " ->"))
    print(f"Execution time: {str(execution_time)[1:-1]}")


if __name__ == "__main__":
    # Place jobs in their respective queues
    for job in jobs:
        if job["queue"] == 1:
            queue1.append(job)
        else:
            queue2.append(job)

    # Displays process, cpu cycle and queue
    print("-"*24)
    print("| {:<5} {:<10} {:<5}".format("PID", "CPU Cycle", "Queue"))
    print("-"*24)
    for job in jobs:
        print(
            "| {:<5} {:<10} {:<5}"
            .format(str(job["id"]), job["cpu cycle"], str(job["queue"]))
        )
    print("-"*24)

    no_movement_btw_queues(copy.deepcopy(queue1), copy.deepcopy(queue2))
    movement_btw_queues(copy.deepcopy(queue1), copy.deepcopy(queue2))
    variable_time_quantum_per_queue(copy.deepcopy(queue1), copy.deepcopy(queue2))
```


```
------------------------
| PID   CPU Cycle  Queue
------------------------
| 1     8          1
| 2     4          2
| 3     1          2
| 4     5          1
| 5     2          2
| 6     3          1
------------------------
No movement between queues: 'P1' -> 'P4' -> 'P6' -> 'P2' -> 'P3' -> 'P5'
Execution time: 0, 8, 13, 16, 20, 21, 23

Movement between queues: 'P1' -> 'P4' -> 'P6' -> 'P2' -> 'P3' -> 'P5' -> 'P1' -> 'P4' -> 'P2' -> 'P1'
Execution time: 0, 3, 6, 9, 12, 13, 15, 18, 20, 21, 23

Variable time quantum per queue: 'P1' -> 'P4' -> 'P6' -> 'P2' -> 'P3' -> 'P5' -> 'P1' -> 'P4'
Execution time: 0, 3, 6, 9, 13, 14, 16, 21, 23
```

DOCUMENTATION

The Multiple Level Queue scheduling algorithm is nothing unique but rather a combination of other process scheduling algorithms. Processes to be executed are placed in different queues based on priority.

```
# NB: Queue 1 has higher priority than Queue 2
# Place jobs in their respective queues
for job in jobs:
    if job["queue"] == 1:
        queue1.append(job)
    else:
        queue2.append(job)
```

The queues go for processing in order of decreasing priority. The processes in their respective queues are then executed using some other process scheduling algorithm.

HOW DOES IT WORK?

There are different ways of executing jobs in their queues but only three methods are considered:

No movement between queues: in this method, queues use First-Come First-Serve (FCFS) logic to choose the process to be executed. In each queue, the processes are loaded into the processor based on the order they arrive i.e. the first process to enter that queue will be the first to be processed and others follow sequentially based on they when arrive.

```
for queue in priority:
        # Execute job in order of arrival (FCFS)
        for job in queue:
            execution_sequence.append(job["id"])
```

Movement between queues: in this method, queues use the Round Robin method to choose the process to be executed. A time quantum is allocated to each process. If a process is unable to finish executing after the time slice has expired, that process is added to next queue to be executed.

```
for job in queue:
    execution_sequence.append(job["id"])
    # Check if job has finished executing
    job["cpu cycle"] -= time_slice
    if job["cpu cycle"] > 0:
        # Check if queue is the last
        if i == len(priority) - 1:
            priority[-1].append(job)
        # Add job to next queue
        else:
            priority[i + 1].append(job)
    else:
        del job
```

Variable time quantum per queue: this method is similar to movement between queues. It also uses the Round Robin method but after iterating through a queue, the time quantum for the next queue doubles. Processes unable to finish executing after expiration of time quantum are added to the next queue which will have a time quantum of double of what it currently is.

```
for job in queue:
    execution_sequence.append(job["id"])
    # Check if job has finished executing
    job["cpu cycle"] -= time_slice
    if job["cpu cycle"] > 0:
        # Check if queue is the last
        if i == len(priority) - 1:
            priority[-1].append(job)
        # Add job to next queue
        else:
            priority[i + 1].append(job)
    else:
        del job
# Double time slice
time_slice += time_slice
```