Task 1

- The process will be running on a general-purpose system.
- Processes are the context associated with a program in execution.
- A process has a life cycle. Once it's finished its execution, it terminates the entire process.
- A process can switch between different stages, such as running, ready, blocked, etc.
- The OS stores a priority queue of process contexts that are ready for execution.
    - I'll use a range of 0 to 32767 for the process ID.
    - Process Allocation Python Code:

```python
class process_allocation:
    def __init__(self):
        self.max_processes = 32768
        self.processes = set()

    def allocate_id(self) -> None:
        for i in range(self.max_processes):
            if i not in self.processes:
                self.processes.add(i)
                print(f"Allocated id {i}")
                return id
        return -1

    def release_id(self, id: int) -> bool:
        if id in self.processes:
            self.processes.remove(id)
            print(f"Released id {id}")
            return True
        return False
```

    - When the program runs out of PIDs, it simply returns -1.
- The context stores information about the process such as:

Process Context Table

Name	**Data Structure Used**
Stack	stack
Heap	heap
Code Area	string (data pointer)
Data Area	string (data pointer)
Register States	array (each index corresponds to a register)
Process ID	int
Parent Process ID	int / None
User ID	int
Current Directory	string
SP	string (data pointer)
PC	string (data pointer)
Priority	int
State	string (ready/blocked/running)

Task 2

State changes.

Current State	Event	New State	Comment
Ready	Reaches top of queue	Running	The process receives it's allocated time slot for the CPU
Running	I/O operation that must be awaited	Blocked	I/O operation could be a printer printing a page.
Blocked	I/O operation finishes	Ready	Printer finishes printing page and process can continue
Running	The process runs out of time on the CPU	Ready	Each queue gets a time quantum, this process runs out.

- Ready refers to the process being ready to be run, however it is waiting to get its timeshare of the CPU.
- Running means, it's currently running on the CPU.
- Blocked means the process has been put on hold until an event completes (generally an I/O operation)

Discuss the mechanism by which an event triggers the change of the state of a process. Consider as examples the switch to the Blocked state, and from Running to Ready. If in the life cycle you defined there isn't the Blocked state, consider another example.

- Say we have a HP application that controls a printer.
- The application will move from the READY state to the RUNNING state when it gets it's allocated time slice on the CPU.
    - Which depends on its priority and the last time it ran.
- If the application needs the printer to print something or needs to connect to the printer, the application will move into a BLOCKED state.
    - In this blocked state, the program is put on a temporary hold on the execution of code until the printing completes.
    - It gets no CPU time until it exits this BLOCKED state.
- Once the printing is completed, the printing application is returned to the READY state.
    - Or possibly directly into the RUNNING state if no other processes are in the queue.

Define the rule by which the priority of a blocked process is increased when I/O is completed and the process becomes ready again.

- Priority-based scheduling (dynamic priorities / multilevel feedback queues) allows a blocked process to receive a boost upon its return to a READY state.
- The longer the process is in a BLOCKED state, the higher its priority boost will be.

- The reasoning behind this priority-based scheduling is to prevent the starvation of other processes and to ensure fairness between processes.

Describe the work (algorithm) carried out by the idle process.

- Many systems have an idle process.
  - This has the lowest priority in the system.
  - When nothing else is there to execute, the CPU switches to the idle process.
- The idle process switches the system to a sleep state.
- The idle process uses the kernel power policy manager.
  - This owns the decision-making and set of rules used to determine the appropriate frequency/voltage.

Task 3

Discuss the role of the scheduler in your system and its principles (e.g., using priority)

- The purpose of the scheduler in a system is to prevent the starvation of processes and to ensure that all processes are treated fairly and given adequate CPU time.
- It will be implemented with **multi-level feedback queues:**
  - 3 queues, Q0, Q1, Q2, with Q0 having the highest priority.
  - Q0 and Q1 will use round-robin scheduling, with time quantum's of 8ms and 16ms respectively.
  - Q2 will use FCFS (first-come, first-serve)
  - Q0 and Q1 are for foreground processes, therefore a starvation-free scheduling algorithm is needed.
  - Q2 will be mainly for background processes, therefore FCFS shouldn't be an issue.

Present the data structures the scheduler is using.

- This scheduler uses an efficient Queue structure while using multi-level feedback queues.
  - This is a Queue implementation from the CS2515 module, that was just used to make the scheduler more efficient.

```
class Queue:

    def __init__(self):
        self.body = [None] * 10
        self.head = 0    #index of first element, but 0 if empty
        self.tail = 0    #index of free cell for next element
        self.size = 0    #number of elements in the queue


    def grow(self):
        oldbody = self.body
        self.body = [None] * (2*self.size)
        oldpos = self.head
        pos = 0
        if self.head < self.tail:     #data is not wrapped around in list
```

```
            while oldpos <= self.tail:
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
        else:                           #data is wrapped around
            while oldpos < len(oldbody):
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
            oldpos = 0
            while oldpos <= self.tail:
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
        self.head = 0
        self.tail = self.size

    def shrink(self):
            oldbody = self.body
            self.body = [None] * math.floor(len(self.body) / 2)
            oldpos = self.head
            pos = 0

            if self.head < self.tail:
                while oldpos <= self.tail:
                    self.body[pos] = oldbody[oldpos]
                    pos += 1
                    oldpos += 1
            else:
                while oldpos < len(oldbody):
                    self.body[pos] = oldbody[oldpos]
                    pos += 1
                    oldpos += 1
                oldpos = 0
                while oldpos <= self.tail:
                    self.body[pos] = oldbody[oldpos]
                    pos += 1
                    oldpos += 1

            self.head = 0
            self.tail = self.size

    def enqueue(self,item):
        # An improved representation would use modular arithmetic
        if self.size == 0:
```

```
            self.body[0] = item      # assumes an empty queue has head at 0
            self.size = 1
            self.tail = 1
        else:
            self.body[self.tail] = item
            # print('self.tail =', self.tail, ': ', self.body[self.tail])
            self.size = self.size + 1
            if self.size == len(self.body):  # list is now full
                self.grow()                  # so grow it ready for next
enqueue
            elif self.tail == len(self.body)-1:  # no room at end, but must be
at front
                self.tail = 0
            else:
                self.tail = self.tail + 1
        #print(self)

    def dequeue(self):
        """ Return (and remove) the item in the queue for longest. """
        # An improved implementation would use modular arithmetic
        if self.size == 0:     # empty queue
            return None
        item = self.body[self.head]
        self.body[self.head] = None
        if self.size == 1:  # just removed last element, so rebalance
            self.head = 0
            self.tail = 0
            self.size = 0
        elif self.head == len(self.body) - 1:  # if head was the end of the
list
            self.head = 0  # we must have wrapped round, so point to start
            self.size = self.size - 1
        else:
            self.head = self.head + 1  # just move the pointer on one cell
            self.size = self.size - 1
        # we haven't changed the tail, so nothing to do

        #if length of internal list is less than 1/4 of the number of items in
the queue then shrink
        if len(self.body) > 10 and self.size < len(self.body)/4:
            self.shrink()

        return item

    def length(self):
        """ Return the number of items in the queue. """
        return self.size
```

- The rest of the scheduler uses an array of these efficient Queues, with the first index of the array being the higher priority queue, and the higher indexes being the lower priorities.
- The scheduler also simulates blocked processes.
    - The “process” class (which is below), will randomly choose whether to become blocked.
    - This is simply to simulate blocked processes and is not how real processes become blocked.
- A process has a 1/10 chance of being blocked every time it runs on the CPU.
    - It then has a 1/5 of being returned to the queue at a higher priority.

Write the scheduler in pseudo-code and explain how it works

```
# 0 = success
# 1 = finished running
# 2 = blocked

class process:
    def __init__(self, pid):
        self.pid = pid
        self.lifespan = uniform(0.1, 2.0)

    def __str__(self):
        return f"({self.pid})"

    # runs for the given time
    def run_for(self, time):
        ### this is to simulate a process being blocked
        chance_of_blocked = randint(0, 10)
        if chance_of_blocked == 1:
            return 2

        self.lifespan -= time
        sleep(time)

        # if process is finished executing
        if self.lifespan <= 0:
            return 1
        # if process has ran out of time
        return 0

class scheduler:
    def __init__(self, amount_of_queues):
        # Initialize a list to hold queues and blocked queue
        self.queues = []
        self.blocked_queue = Queue()
```

```
        for i in range(amount_of_queues):
            self.queues.append(Queue())

        # current number of processes
        self.amount_of_processes = 0

        self.initial_quantum = 0.1

    def _print_queues(self):
        for queue in range(len(self.queues)):
            print(f"Queue {queue+1} : {self.queues[queue]}")
        print(f"Blocked Queue: {self.blocked_queue}")
        print("------------------")

    #adds a process to the specified priority queue
    def add_process(self, process, priority):
        self.queues[priority].enqueue(process)
        self.amount_of_processes += 1

    # runs the next process in the given priority queue
    def run_process(self, priority):
        # get the process next in queue or randomly from the blocked queue
            # this is only to simulate blocked processes randomly returning
        random_number = randint(1, 5)

        if random_number == 1 and self.blocked_queue.length() > 0:
            current_process = self.blocked_queue.dequeue()
            priority = -1
        else:
            current_process = self.queues[priority].dequeue()

        # if priority at the lowest priority queue
        if priority < len(self.queues)-1:
            new_priority = priority + 1
            quantum = self.initial_quantum * (priority+1)
        else:
            new_priority = priority
            quantum = current_process.lifespan

        result = current_process.run_for(quantum)
        self.amount_of_processes -= 1

        if result == 0:
            self.add_process(current_process, new_priority)
        elif result == 2:
            self.blocked_queue.enqueue(current_process)
            self.amount_of_processes -= 1
```

```
    # run next highest priority process
    def run_next_process(self):
        for i in range(len(self.queues)):
            if self.queues[i].length() > 0:
                self.run_process(i)
                break


    def start_cpu(self):
        while self.amount_of_processes > 0:
            # print the contents of the queues
            self._print_queues()
            # execute next process
            self.run_next_process()
        self._print_queues()


s = scheduler(10)
p = process_allocation()
for i in range(20):
    s.add_process(process(p.allocate_id()), randint(0, 3))
s.start_cpu()
```

- The scheduler initialises by creating all the Queues needed for the multi-level feedback queue algorithm.
    - It also creates a blocked queue, which will consist of blocked processes.
- It also initialises the time quantum for the highest priority queue.
- The “run_process” function gets the next process in the given priority level.
    - It also simulates a blocked process, by randomly choosing whether to take from the blocked queue instead of the highest priority queue.
- The “run_next_process” simply runs through each queue and finds the highest priority process, which is what will be ran in the “run_process” function.
- Then the “start_cpu” function checks if there are processes still to be ran and continues to run through the queues.
- If process finishes its execution (by running down it’s clock in the process class), it is removed from the multi-level feedback queues.
- The “_print_queues” function is simply a helper function to see what’s occurring in the queues.