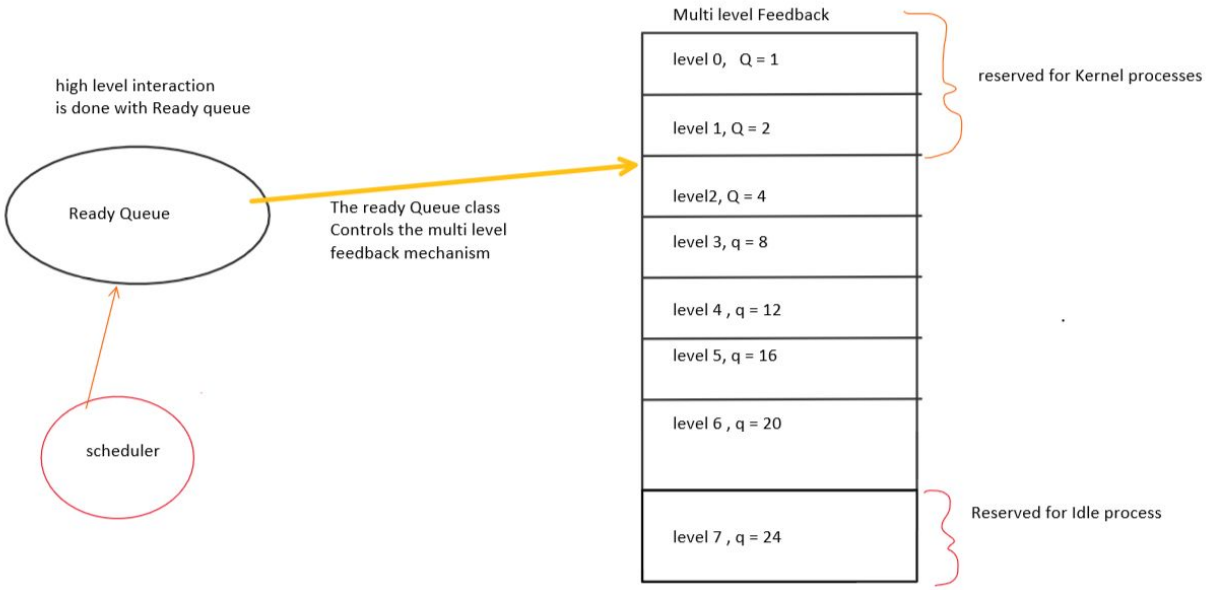
Michal Wolas 117308883

Task 1



When all user processes are terminated, level 7 is accessed where the idle process is ran.

The idle process, provided there is nothing in the blocked queue, will begin to apply power saving policy.
It will perform tasks like setting frequency / voltage operating state, utilizing the processor, applying battery saving mode if applicable and other user specific power policies. Once this is done the system is idle.

idle process :
stage 1 : setting frequency / voltage operating state
stage 2 : applying custom User power saving policy
stage 3 : Utilizing the process
stage 4 : applying battery saving mode

Task 2

Class Process: holds the necessary information about processes
Attributes - id, name, priority, execTime, numOfIOs, offset
kernel True/False # determines if it can be added at level 0 or 1
Def setOffset - sets the offset attribute

class ReadyQueue:

the ready class will contain and handle the multi level feedback queue.
a self variable called, mutilevelfeedbackqueue, is a dictionary where the keys are the levels and the values are the equivalent lists, all of which are using a round robin style operation.
If a process at level 6 is blocked, when it becomes unblocked it will get one more level boost as opposed to a single level boost. level 6 --> level 4.

attributes - multilevelfeedbackqueue # dictionary key = level, value = queue

def addReadyProcess(process):
add the process at the processes priority level.
perform necessary check to make sure the process is on the correct level.
0 and 1 are reserved for kernel processes, 7 is reserved for idle process.

def nextProcess():
gets the next process by iterating over the dictionary, and popping the highest priority process from a level using round robin technique(if there is multiple processes in that level.

def is_empty():
reports if a given list is empty or not
def empty():
reports if the whole ready queue is empty or not, ignores idle process.

Class Scheduler:
uses the two queues, and a stack , to control the flow of the execution. Handles the interruptions and blocks of processes. Everything is controlled through this class. It also terminates process and randomly triggers interrruptions. Takes care of generating random offsets for I/O processes to be "completed"

Attributes - CurrentlyRunning, numOfCycles, interruptProcess
Has_a (by using composition) - readyQueue, blockedQueue, interruptStack

Def schedule(interruption=False):
if theres no user processes in any queue:
run idle process
call cleanUp()
If interruption:
handle the interruption
else If a process was running, seize its execution,

If it was interruption process:
Say interruption completed
Resume the process from the stack
Else:
at the start of each timeslice the previous process is updated
update it, place it in the ready queue or terminate it.

Get the next process from ready queue and run it.

If that process needs an I/O , call blockProcess.
timeslice ended

def cleanUp():
this function performs the task of applying Power saving policies,
it is activated by the idle process only when there is no user processes in ready/blocked queue.

Def blockProcess(process):
Seizes the process from running, blocks it.
Call generateOffset on that process
Places it in the blockedQueue.
Gets the next process from the readyQueue and executes it.

Def terminate(process):
Deletes the process instance

def completedIO(self, process):
# called by test function to say that an IO was completed
removes a process from the blocked queue and adds it to the ready queue
self.readyQueue.addReadyProcess(self.blockedQueue._queue[process])

Def generateOffset(process):
Randomizes the wait time for the I/O completion
It will return to ready queue in a random number of clock cycles.
Process.offset = self.numOfcyclses + random integer

Def interruption:
Random chance of an interruption happening,
cannot interrupt if Kernel process is running.
If it is it makes the current process go to the suspended stack.
Calls schedule informing that interruption happened
Schedule(True)

Class BlockedQueue: data structure, adding/removing functionality
Attributes - queue= []

Def addBlockedProcess(process):
Append it to the queue

Def IOCompletion(process):

Remove the process from the queue
Queue.pop(queue.index(process))

Class InterruptStack: data structure, adding/removing functionality
Attributes - stack= []

Def addSuspendedProcess(proces)
Append it to the list

Def getSuspendedProcess
Return Stack.pop()

class test:
this class is used to test the scheduler, it manages its instance and the IO completetions as an outside factor.

attributes - scheduler instance , list of processe

createScheduler():
creates an instance of the scheduler by populating it with provided processes

triggerIOCompletetion():
by comparing the offsets of all processes in the blocked queue, that are given to a blocked process, with schedulers clock cycles, it sends scheduler messages when a process has completed its IO
by calling schedulers method completedIO(process)

runScheduler():
this is ran periodically, this method calls scheduler.schedule()
and triggerIOCompletetion

Task 3

```
import time
import sched
import random


class Process:  # Each process is an instance of this class
   def __init__(self, id, name, priority, execTime, numOfIOs, kernel=False):  # all attributes visible to
every class
       self.id = id
       self.name = name
       self.priority = priority  # integer number that represents the level of the queue
       self.execTime = execTime  # integer number of the amount of Time slices it requires in total.
       self.numOfIOs = numOfIOs  # total int number of I/O operations required, in sequence.
```

```
        self.state = 1  # 0 = Blocked, 1 = Running, 2 = Waiting, 3 is suspended , 4 is terminated
        self.offset = 0  # offset used to indicate how much time it will take for its IO to come back
        self.kernel = kernel  # if the process is a kernel process or not

    def setOffset(self, offset):  # sets the IO offset when this process instance triggers IO
        self.offset = offset

# Data structure and method to control and use Multi-level feedback queue.
class ReadyQueue:
    def __init__(self):
        self._multiLevelFeedback = {0: [], 1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: []}
        self.idle = Process("idle", "idle", 7, 100, 0)

    # adds the process into the appropriate queue
    # it looks at whether it is a Kernel process or not, or if it went outside its boundries.
    def addReadyProcess(self, process):

        process.state = 2 # update state to waitin
        if process.priority == 7:  # level 7 reserved for idle process
            process.priority = 6
        if process.kernel:
            if process.priority > 1:
                process.priority = 1
        else:
            if process.priority <= 1:
                process.priority = 2  # put back to level 2 if not kernel

        self._multiLevelFeedback[process.priority].append(process)

    # gives the next ready process of highest priority
    def NextProcess(self):
        for level in self._multiLevelFeedback:
            if not self.is_empty_level(self._multiLevelFeedback[level]):
                return self._multiLevelFeedback[level].pop(0)
        return self.idle

    def is_empty_level(self, list):  # reports if a list(queue) is empty or not
        if len(list) == 0:
            return True
        return False

    def empty(self):  # reports if the ready queue is empty or not
        for level in self._multiLevelFeedback:
```

```
            if not self.is_empty_level(self._multiLevelFeedback[level]):
                return False
        return True

    def __str__(self):  # prints the list of processes using names of processes
        output = ""
        for level in self._multiLevelFeedback:
            output += "level %d:  " % (level)
            for processes in self._multiLevelFeedback[level]:
                output += " %s, " % (processes.name)
            if level == 7:
                output += " Idle process, "
            output = output[:-2]
            output += "\n"
        return output
class BlockedQueue():
    # handles the blocked queue, restoration of processes into ready queue by receiving I/O completion
triggers
    def __init__(self):
        self._queue = []

    def block(self, process):
        # blocks the given process by placing it into the blockedQueue
        # returns the next process to take control of the CPU
        process.numOfIOs -= 1
        self._queue.append(process)

    def IOCompletion(self, process):  # triggered by a test function
        # I/O for a specific process is completed
        # find and remove that process form the blockedQueue
        self._queue.pop(self._queue.index(process))

    def empty(self):  # reports if the blocked queue is empty or not
        if len(self._queue) == 0:
            return True
        return False

    def __str__(self):  # prints the list of processes using names of processes
        if len(self._queue) == 0:
            return "is Empty"
        processList = ""
        for x in range(len(self._queue)):
            processList += self._queue[x].name + ", "
```

```
        return processList[:-2]
class InterruptStack:
    # data structure for the stack for Suspended Processes
    def __init__(self):
        self.stack = []

    def addSuspendedProcess(self, process):  # adds a process to suspended stack
        self.stack.append(process)

    def getSuspendedProcess(self):  # gets the top element from the stack
        return self.stack.pop()

# responsible for control Ready, Blocked and interrupt data structures to schedule processes.
# is able to block and suspend processes as well as make them run
class Scheduler:

    # initialising the data structures and necessary variables/lists
    def __init__(self):
        self.readyQueue = ReadyQueue()
        self.blockedQueue = BlockedQueue()
        self.interruptStack = InterruptStack()
        self.currentlyRunning = [None]
        self.interruptProcess = Process("Interruption", "interruption", 1, 2, 0)
        self.numOfCycles = 0
        self._quanta = {0: 1, 1: 2, 2: 4, 3: 8, 4: 12, 5: 16, 6: 20, 7: 24}  # predefined quanta values for
each level
        self.powerSavingPolicy = "stage1"

    def add(self, process):  # adds processes to the schedulers queue , to be used outside the class
        self.readyQueue.addReadyProcess(process)

    # triggered by idle process, runs the power saving policy.
    def cleanUp(self):
        # the clean up policy for saving power.
        if self.powerSavingPolicy == "stage1":
            print("There is no processes in the Ready/Blocked queue, Idle Process is executing Power saving
policy: ")
            print("stage 1 : setting frequency / voltage operating state")
            self.powerSavingPolicy = "stage2"

        elif self.powerSavingPolicy == "stage2":
            print("stage 2 : applying custom User power saving policy")
```

```
            self.powerSavingPolicy = "stage3"

        elif self.powerSavingPolicy == "stage3":
            print("stage 3 : Utilizing the process")
            self.powerSavingPolicy = "stage4"

        elif self.powerSavingPolicy == "stage4":
      print("stage 4 : applying battery saving mode")
            self.powerSavingPolicy = "stage5"
        else:
            print("System is idle")

    # manages blocks, interrupts and process scheduling
    # it can terminate and put the process back into ready queue as necessary
    def schedule(self, interrupt=False, blockOccured=False):
        # called every time slice, to run a process and update it when its finished
        if self.readyQueue.empty() and self.currentlyRunning[0].id == "idle" and self.blockedQueue.empty():
            self.cleanUp()
        else:
            if self.powerSavingPolicy != "stage1":
                self.powerSavingPolicy = "stage1"

            if self.currentlyRunning[0] != None and not interrupt and not blockOccured:
                # to ensure there is a valid process running so it can be updated
                quanta = self._quanta[self.currentlyRunning[0].priority]  # get the value to decrement from
time slice
                self.currentlyRunning[0].execTime -= quanta
                if self.currentlyRunning[0].execTime <= 0:  # when the process is finished
                    self.terminate(self.currentlyRunning[0])
                else:
                    if self.currentlyRunning[0].id != "idle":
                        self.currentlyRunning[0].priority += 1
                        self.readyQueue.addReadyProcess(self.currentlyRunning[0])

            # schedule next process to run
            if not blockOccured and not interrupt:
                if self.currentlyRunning[0] == None:

                    self.currentlyRunning[0] = self.readyQueue.NextProcess()

                elif self.currentlyRunning[0].id == "Interruption":
                    self.currentlyRunning[0] = self.interruptStack.getSuspendedProcess()
                    print("Process %s has been resumed from the stack" % (self.currentlyRunning[0].name))
```

```
                else:
                    self.currentlyRunning[0] = self.readyQueue.NextProcess()

            elif interrupt:
                self.currentlyRunning[0] = self.interruptProcess
            print("Currently running: process: %s priority level: %d" % (
                self.currentlyRunning[0].name, self.currentlyRunning[0].priority))

            if self.currentlyRunning[0].name != "interruption" and self.currentlyRunning[0].name != "idle":
                # queue isn't altered with either idle or interrupt processes
                print("Ready Queue while running %s :\n%s " % (self.currentlyRunning[0].name,
self.readyQueue))

            # check if an I/O is necessary
            if self.currentlyRunning[0].numOfIOs != 0:
                self.blockProcess(self.currentlyRunning[0])

            if not self.readyQueue.empty() and not blockOccured:
                self.interrupt()  # every time slice there is a chance for interruption to occur

        if not blockOccured and not interrupt:  # as it blocking / new processes taking control happen in
the same time slice
            print("\n-------------- Time slice Completed , %s updated ------------------\n" % (
                self.currentlyRunning[0].name))
            self.numOfCycles += 1

    # called by schedule class ( to extract functionality )
    # blocks a process, places it in the blocked queue, and updates its priority
    def blockProcess(self, process):

        self.generateOffset(process)
        process.state = 0
        print("\tProcess %s is performing an I/O and is BLOCKED" % (process.name))
        if process.priority == 6:
            process.priority -= 1  # when process is at the lowest level possible, increase its priority
more
        process.priority -= 1  # increase the process' priority
        self.blockedQueue.block(process)
        self.currentlyRunning[0] = self.readyQueue.NextProcess()
        print("\t\tCPU is free and process %s will take control\n" % (self.currentlyRunning[0].name))
        self.schedule(False, True)

    # triggered by a test function ( "accepts IO completetion ")
```

```
    # places the process from the blocked queue to the ready queue
    def completedIO(self, process):  # called by test function to say that an IO was completed
        print(" I/O for Process %s is done, process is now placed in the Ready Queue:" % (
            self.blockedQueue._queue[process].name))
        self.readyQueue.addReadyProcess(self.blockedQueue._queue[process])
        self.blockedQueue.IOCompletion(self.blockedQueue._queue[process])
        print(self.readyQueue)

    # called every time slice, for a chance of an interrupt happening
    # suspends currently running process and makes the schedule() handle the interruption
    def interrupt(self):
        # interrupts occur randomly
        chance = random.randint(0,5)
        if chance == 2:
            if self.currentlyRunning[0].id != "Interruption":
                if not self.currentlyRunning[0].kernel:  # CANNOT INTERRUPT A KERNEL PROCESS
                    print("INTERRUPTION \n")
                    # place the interrupted process on the stack
                    self.currentlyRunning[0].state = 3  # 3 = suspended
                    self.interruptStack.addSuspendedProcess(self.currentlyRunning[0])
                    self.schedule(True)  # force the execution of the interrupt

    # terminates a processm invoked by schedule()
    def terminate(self, process):
        process.staet = 4  # terminated = 4
        print("------------\nProcess %s has been Completed \n-----------\n" % (process.name))

    # random number, signifying in how many time slices a blocked process will come back in.
    def generateOffset(self, process):  # a random number , it determines when a process is returned
        offset = random.randint(2, 4) + self.numOfCycles
        process.setOffset(offset)  # if a process has I/O and it triggers it it will get an Offset


# test class capable of running schedule
class Test():
    def __init__(self, processList):
        self.processList = processList
        self.scheduler = Scheduler()

    # creates the scheduler by adding processes
    def createScheduler(self):  # build the ready Queue of the scheduler instance
        for process in self.processList:
            self.scheduler.add(process)
```

```
    def runScheduler(self):  # this method is ran using "import sched" event scheduling

        self.scheduler.schedule()
        self.triggerIOCompleteion()

    def triggerIOCompleteion(self):  # this goes through the blocked list
        # and checks a blocked process offset has reached the num of cycles
        # it then restores the process
        for process in range(len(self.scheduler.blockedQueue._queue)):
            if self.scheduler.blockedQueue._queue[process].offset < self.scheduler.numOfCycles:
                self.scheduler.completedIO(process)
                break  # only one restore possible per TimeSlice


# id , name , priority execTime, I/O, kernel T/F
A = Process("A", "A", 3, 15, 1)
B = Process("B", "B", 4, 7, 0)
C = Process("C", "C", 6, 5, 1)
D = Process("D", "D", 2, 14, 0)
E = Process("E", "E", 5, 9, 1)
F = Process("F", "F", 2, 3, 2)
G = Process("G", "G-kern", 0, 1, 0, True)
H = Process("Kernel", "kernel", 0, 2, 0, True)
I = Process("I", "I", 4, 18, 0)
myProcessList = [A, B, C, D, E, F, G, H, I]

TestScheduler = Test(myProcessList)  # initiate the test class with process to run
TestScheduler.createScheduler()  # create a ready to execute version of scheduler to test

# Depending on the version of Python, this might not work
timeSlice = sched.scheduler(time.time, time.sleep)
while True:  # infinite loop to run the scheduler
    # every 3 seconds call  TestScheduler.runScheduler , which is the testing Scheduler class
    timeSlice.enter(3, 1, TestScheduler.runScheduler, ())
    timeSlice.run()
```

Task 4

```
Currently running: process: G-kern priority level: 0
Ready Queue while running G-kern :
level 0:   kernel
level 1:
level 2:   D,  F
level 3:   A
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process


-------------- Time slice Completed , G-kern updated ------------------

------------
Process G-kern has been Completed
-----------

Currently running: process: kernel priority level: 0
Ready Queue while running kernel :
level 0:
level 1:
level 2:   D,  F
level 3:   A
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process


-------------- Time slice Completed , kernel updated ------------------
```

```
Currently running: process: kernel priority level: 1
Ready Queue while running kernel :
level 0:
level 1:
level 2:   D,  F
level 3:   A
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process


-------------- Time slice Completed , kernel updated ------------------

------------
Process kernel has been Completed
-----------

Currently running: process: D priority level: 2
Ready Queue while running D :
level 0:
level 1:
level 2:   F
level 3:   A
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process


-------------- Time slice Completed , D updated ------------------
```

```
Currently running: process: F priority level: 2
Ready Queue while running F :
level 0:
level 1:
level 2:
level 3:   A,  D
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process

    Process F is performing an I/O and is BLOCKED
        CPU is free and process A will take control

Currently running: process: A priority level: 3
Ready Queue while running A :
level 0:
level 1:
level 2:
level 3:   D
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process

    Process A is performing an I/O and is BLOCKED
        CPU is free and process D will take control

Currently running: process: D priority level: 3
Ready Queue while running D :
level 0:
level 1:
level 2:
level 3:
level 4:   B,  I
level 5:   E
level 6:   C
level 7:   Idle process
```

```
-------------- Time slice Completed , D updated ------------------

Currently running: process: B priority level: 4
Ready Queue while running B :
level 0:
level 1:
level 2:
level 3:
level 4:   I,  D
level 5:   E
level 6:   C
level 7:   Idle process

INTERRUPTION

Currently running: process: interruption priority level: 1

-------------- Time slice Completed , interruption updated ------------------

------------
Process interruption has been Completed
-----------

Process B has been resumed from the stack
Currently running: process: B priority level: 4
Ready Queue while running B :
level 0:
level 1:
level 2:
level 3:
level 4:   I,  D
level 5:   E
level 6:   C
level 7:   Idle process
```

… operation continues as normal ….

When we are waiting for an IO to complete Idle process is running, but its waiting (not applying the Power saving mode) as there are processes in the blocked queue.

```
-------------- Time slice Completed , F updated ------------------

------------
Process F has been Completed
-----------

Currently running: process: C priority level: 6
Ready Queue while running C :
level 0:
level 1:
level 2:
level 3:
level 4:
level 5:
level 6:
level 7:   Idle process

    Process C is performing an I/O and is BLOCKED
        CPU is free and process idle will take control

Currently running: process: idle priority level: 7

-------------- Time slice Completed , idle updated -------------------

Currently running: process: idle priority level: 7

-------------- Time slice Completed , idle updated -------------------
```

When all the processes finish …

```
------------
Process C has been Completed
-----------

Currently running: process: idle priority level: 7

-------------- Time slice Completed , idle updated ------------------

There is no processes in the Ready/Blocked queue, Idle Process is executing Power saving policy:
stage 1 : setting frequency / voltage operating state

-------------- Time slice Completed , idle updated ------------------

stage 2 : applying custom User power saving policy

-------------- Time slice Completed , idle updated ------------------

stage 3 : Utilizing the process

-------------- Time slice Completed , idle updated ------------------

stage 4 : applying battery saving mode

-------------- Time slice Completed , idle updated ------------------

System is idle

-------------- Time slice Completed , idle updated ------------------

System is idle

-------------- Time slice Completed , idle updated ------------------

System is idle
```