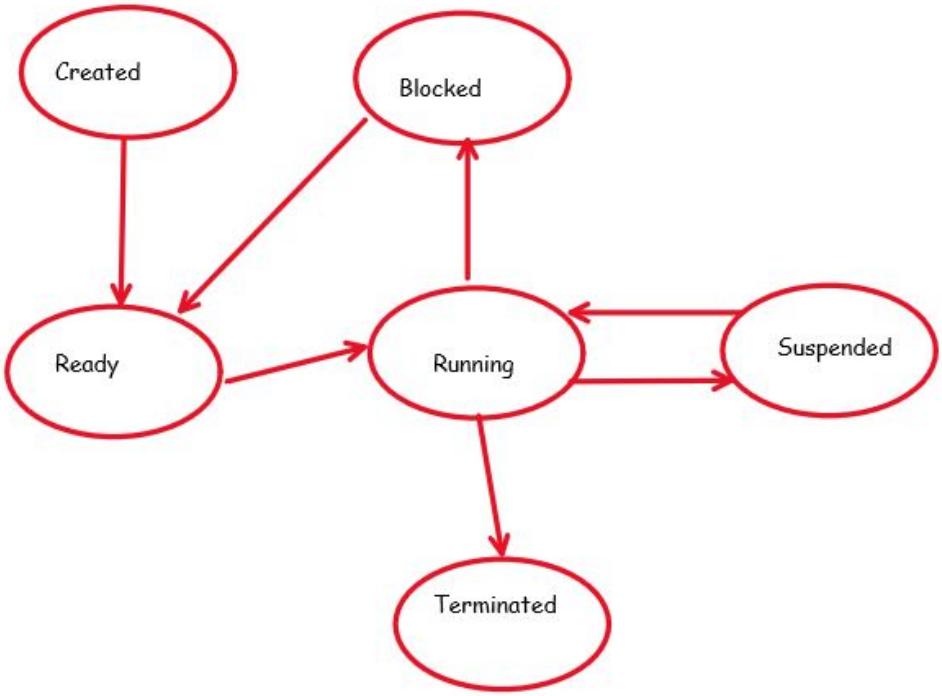

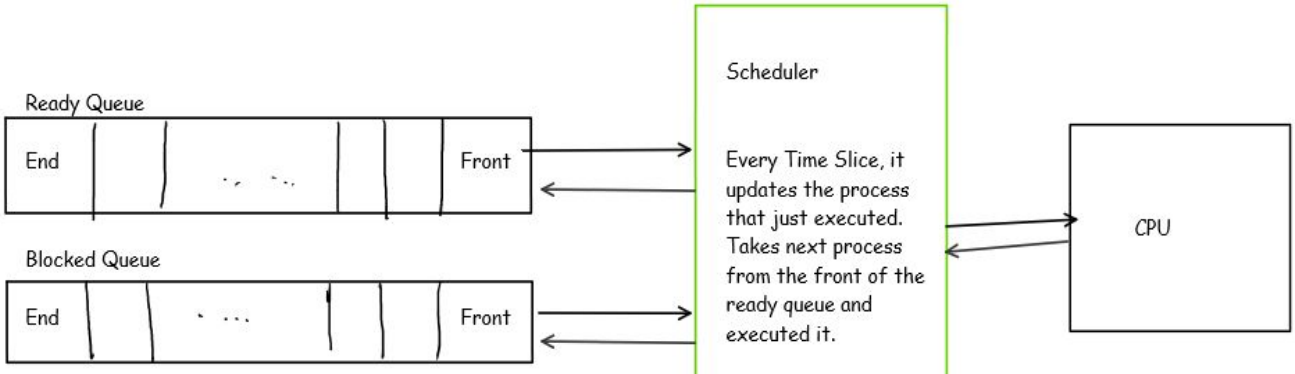

1. Process is executed for the time slice, updated and placed into the ready queue
2. Process is executed for the time slice, updated and terminated if finished.
3. Process is executed and I/O occurs. It is blocked and placed into the blocked queue, next process is taken from Ready queue to be executed
4. A process can be interrupted, it is paused in the stack

Event	Reaction	Comment
Creation	Process is added, asking to join the ready queue	
Waiting	Process is in the ready Queue, waiting to be executed	
Running	Is executed by the CPU	
I/O request	Process is placed in the blocked queue	can only happen when it is executing
I/O response	Placed into ready q , removed from blocked q	After the response arrives
Interruption	Placed onto suspended stack	Can resume after interruption is handled
Termination	Removed the process from cycle	Process is deleted

What is the difference between the blocked and the suspended transitions of a process?

When a process is blocked, Its journey goes from the blocked queue to the ready queue. It then has to wait for the CPU time.
When a process is suspended it is 'paused' and placed into a stack. The interruption is handled. Then the process is taken from the stack and is executed, it does not have to go into the ready queue like the blocked processes.

2. Design

Pseudo-code
Because of has_a relationships, I cannot add the process from the blocked class into the ready Queue. Has to be managed by the scheduler / class that is testing it
Classes - Scheduler , Process, ReadyQueue, BlockedQueue, InterruptStack

Class Process: holds the necessary information about processes
Attributes - id, name, execTime, numOfIOs, offset

Def setOffset - sets the offset attribute

Class ReadyQueue: data structure, adding/removing functionality

Attributes - queue = [] , idleProcess

Def addReadyProcess(process)
Appends the process to the queue

Def nextProcess(): returns and removes the next element in the queue

Return queue.pop(0)

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 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 interruptions. 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 interruption:
handle the interruption
If a process was running, seize its execution,
If it was interruption process:
Say interruption completed
Resume the process from the stack
Else:
 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.

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 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,
If it is it makes the current process go to the suspended stack.
Calls schedule informing that interruption happened
Schedule(True)

3. Python Code

```
import random
import time
import sched
# Each process is an instance of this class
class Process:
    def __init__(self, id, name, execTime, numOfIOs):  # all attributes visible to every class
        self.id = id
        self.name = name
        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

    def setOffset(self, offset):
        self.offset = offset



class ReadyQueue:
    # handles the ready queue using round-robin method, provides the next process to run for the
scheduler class
    def __init__(self):
        self._queue = []
        self._idle = Process("idle", "idle", 1, 0)
        # create an idle process to be ran when readyQueue is empty.
```

```
    def addReadyProcess(self, process):  # adds a process to the queue
        process.state = 2 # set the process state to ready = 2
        self._queue.append(process)
        # print("Process '%s' added to a queue of %d" % (process.name,len(self._queue) - 1))

    def NextProcess(self):  # removes and returns the process that has been there the longest
        if len(self._queue) < 1:
            return self._idle
        proc =  self._queue.pop(0)
        proc.state = 1
        return proc

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

        return processList[:-2] + " <-end"


class BlockedQueue():
    # handles the blocked queue, adding and removing processes from this data structure
    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 __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):
        self.stack.append(process)

    def getSuspendedProcess(self):
        return self.stack.pop()


class Scheduler:
    def __init__(self):
        self.readyQueue = ReadyQueue()
        self.blockedQueue = BlockedQueue()
        self.interruptStack = InterruptStack()
        self.currentlyRunning = [None]
        self.interruptProcess = Process("Interruption", "interruption", 1, 0)
        self.numOfCycles = 0

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

    def schedule(self, interrupt=False, blockOccured=False):
        # called every time slice, to run a process and update it when its finished
        if len(self.readyQueue._queue) == 0 and self.currentlyRunning[0].id == "idle":
            print("Processor is idle")
        else:
            if self.currentlyRunning[0] != None and not interrupt and not blockOccured:
                # to ensure there is a valid process running so it can be updated
                self.currentlyRunning[0].execTime -= 1
                if self.currentlyRunning[0].execTime <= 0:  # when the process is finished
                    self.terminate(self.currentlyRunning[0])
```

```
                else:
                    if self.currentlyRunning[0].id != "idle":
                        self.readyQueue.addReadyProcess(self.currentlyRunning[0])

            print("Ready Queue : %s" % (self.readyQueue))

            # 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 " % (self.currentlyRunning[0].name))
            print("Ready Queue : %s \n" % (self.readyQueue))

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

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

            if len(self.readyQueue._queue) != 0 and not blockOccured:
                self.interrupt()  # every time slice there is a chance for interruption to occur

    def blockProcess(self, process):
        self.generateOffset(process)
        process.state = 0
        print("\tProcess %s is performing an I/O and is BLOCKED" % (process.name))
        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)

    def interrupt(self):
        # interrupts occur randomly
        chance = random.randint(0, 5)
        if chance == 2:
            if self.currentlyRunning[0].id != "Interruption":
                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

    def terminate(self, process):
        process.staet = 4 # terminated = 4
        print("Process %s has been Completed" % (process.name))

    def generateOffset(self, process):  # a random number , it determines when a process is returned
        offset = random.randint(2, 5) + self.numOfCycles
        process.setOffset(offset)  # if a process has I/O and it triggers it it will get an Offset


class Test():
    def __init__(self, processList):
        self.processList = processList
        self.scheduler = Scheduler()

    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 number 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:
                print(" I/O for Process %s is done, process is now placed in the Ready Queue \n" %
 (self.scheduler.blockedQueue._queue[process].name))
                # as an outer( not in scheduler class ) function triggered the IO completion
                # it is added back to the ready queue by calling on the schedulers' ready queue
 method.
                self.scheduler.readyQueue.addReadyProcess(self.scheduler.blockedQueue._queue[process])
                self.scheduler.blockedQueue.IOCompletion(self.scheduler.blockedQueue._queue[process])
                break  # only one restore possible per TimeSlice

        #   id   name, execTime, I/O
 A = Process("A", "A",   2,       0)
 B = Process("B", "B", 2, 1)
 C = Process("C", "C", 3, 0)
 D = Process("D", "D", 1, 1)

 myProcessList = [A, B, C, D]

 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, ())
imeSlice.run()

 # if sched library doesn't work use the below

 # TestScheduler.runScheduler()
 # TestScheduler.runScheduler()
 # TestScheduler.runScheduler()
 # TestScheduler.runScheduler()
 # TestScheduler.runScheduler()
 # TestScheduler.runScheduler()
```

4. Testing

```
Ready Queue : Front-> A, B, C, D <-end
Currently running: process A
Ready Queue : Front-> B, C, D <-end

-------------- Time slice Completed , A updated ------------------

INTERRUPTION

Ready Queue : Front-> B, C, D <-end
Currently running: process interuption
Ready Queue : Front-> B, C, D <-end

-------------- Time slice Completed , interuption updated -----------

Process interuption has been Completed
Ready Queue : Front-> B, C, D <-end
Process A has been resumed from the stack
Currently running: process A
Ready Queue : Front-> B, C, D <-end

-------------- Time slice Completed , A updated ------------------

Ready Queue : Front-> B, C, D, A <-end
Currently running: process B
Ready Queue : Front-> C, D, A <-end

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

Ready Queue : Front-> D, A <-end
Currently running: process C
Ready Queue : Front-> D, A <-end

-------------- Time slice Completed , C updated ------------------
```

```
Ready Queue : Front-> D, A, C <-end
Currently running: process D
Ready Queue : Front-> A, C <-end

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

Ready Queue : Front-> C <-end
Currently running: process A
Ready Queue : Front-> C <-end

-------------- Time slice Completed , A updated ------------------

Process A has been Completed
Ready Queue : Front-> C <-end
Currently running: process C
Ready Queue : is Empty

-------------- Time slice Completed , C updated ------------------

 I/O for Process B is done, process is now placed in the Ready Queue

Ready Queue : Front-> B, C <-end
Currently running: process B
Ready Queue : Front-> C <-end

-------------- Time slice Completed , B updated ------------------

 I/O for Process D is done, process is now placed in the Ready Queue

Ready Queue : Front-> C, D, B <-end
Currently running: process C
Ready Queue : Front-> D, B <-end

-------------- Time slice Completed , C updated ------------------
```

```
Process C has been Completed
Ready Queue : Front-> D, B <-end
Currently running: process D
Ready Queue : Front-> B <-end

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

Process D has been Completed
Ready Queue : Front-> B <-end
Currently running: process B
Ready Queue : is Empty

-------------- Time slice Completed , B updated ------------------

Process B has been Completed
Ready Queue : is Empty
Currently running: process idle
Ready Queue : is Empty

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

Processor is idle
-------------- Time slice Completed , idle updated ------------------

Processor is idle
-------------- Time slice Completed , idle updated ------------------

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