def main():

    # Taking the number of processes

    n = int(input("Enter number of process: "))

    # Matrix for storing Process Id, Burst Time, Average Waiting Time & Average Turn Around Time.

    A = [[0 for j in range(4)] for i in range(100)]

    total, avg\_wt, avg\_tat = 0, 0, 0

    print("Enter Burst Time:")

    for i in range(n): # User Input Burst Time and alloting Process Id.

        A[i][1] = int(input(f"P{i+1}: "))

        A[i][0] = i + 1

    for i in range(n): # Sorting process according to their Burst Time.

        index = i

        for j in range(i + 1, n):

            if A[j][1] < A[index][1]:

                index = j

        temp = A[i][1]

        A[i][1] = A[index][1]

        A[index][1] = temp

        temp = A[i][0]

        A[i][0] = A[index][0]

        A[index][0] = temp

    A[0][2] = 0

    for i in range(1, n):

        A[i][2] = 0

        for j in range(i):

            A[i][2] += A[j][1]

        total += A[i][2]

    avg\_wt = total / n

    total = 0

    # Calculation of Turn Around Time and printing the data.

    # P = Processes , BT = Burst time , WT = Waiting time , TAT = Turn around time.

    print("P     BT  WT  TAT")

    for i in range(n):

        A[i][3] = A[i][1] + A[i][2]

        total += A[i][3]

        print(f"P{A[i][0]}   {A[i][1]}   {A[i][2]}   {A[i][3]}")

    avg\_tat = total / n

    print(f"Average Waiting Time= {avg\_wt}")

    print(f"Average Turnaround Time= {avg\_tat}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

def findWaitingTime(processes, n, wt):

    wt[0] = 0

    # calculating waiting time

    for i in range(1, n):

        wt[i] = processes[i - 1][1] + wt[i - 1]

# Function to calculate turn around time

def findTurnAroundTime(processes, n, wt, tat):

    # Calculating turnaround time by

    # adding bt[i] + wt[i]

    for i in range(n):

        tat[i] = processes[i][1] + wt[i]

def findavgTime(processes, n):

    wt = [0] \* n

    tat = [0] \* n

    # Function to find waiting time

    # of all processes

    findWaitingTime(processes, n, wt)

    # Function to find turn around time

    # for all processes

    findTurnAroundTime(processes, n, wt, tat)

    # Display processes along with all details

    print("\nProcesses Burst Time Waiting",

        "Time Turn-Around Time")

    total\_wt = 0

    total\_tat = 0

    for i in range(n):

        total\_wt = total\_wt + wt[i]

        total\_tat = total\_tat + tat[i]

        print(" ", processes[i][0], "\t\t",

            processes[i][1], "\t\t",

            wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f " % (total\_wt / n))

    print("Average turn around time = ", total\_tat / n)

def priorityScheduling(proc, n):

    # Sort processes by priority

    proc = sorted(proc, key=lambda proc: proc[2],

                reverse=True)

    print("Order in which processes gets executed")

    for i in proc:

        print(i[0], end=" ")

    findavgTime(proc, n)

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # Process id's

    proc = [[1, 10, 1],

            [2, 5, 0],

            [3, 8, 1]]

    n = 3

    priorityScheduling(proc, n)

def findWaitingTime(processes, n, bt,

                        wt, quantum):

    rem\_bt = [0] \* n

    # Copy the burst time into rt[]

    for i in range(n):

        rem\_bt[i] = bt[i]

    t = 0 # Current time

    while(1):

        done = True

        for i in range(n):

            if (rem\_bt[i] > 0) :

                done = False # There is a pending process

                if (rem\_bt[i] > quantum) :

                    t += quantum

                    rem\_bt[i] -= quantum

                else:

                    t = t + rem\_bt[i]

                    wt[i] = t - bt[i]

                    rem\_bt[i] = 0

        # If all processes are done

        if (done == True):

            break

# Function to calculate turn around time

def findTurnAroundTime(processes, n, bt, wt, tat):

    # Calculating turnaround time

    for i in range(n):

        tat[i] = bt[i] + wt[i]

def findavgTime(processes, n, bt, quantum):

    wt = [0] \* n

    tat = [0] \* n

    findWaitingTime(processes, n, bt,

                        wt, quantum)

    findTurnAroundTime(processes, n, bt,

                                wt, tat)

    # Display processes along with all details

    print("Processes Burst Time  Waiting",

                    "Time Turn-Around Time")

    total\_wt = 0

    total\_tat = 0

    for i in range(n):

        total\_wt = total\_wt + wt[i]

        total\_tat = total\_tat + tat[i]

        print(" ", i + 1, "\t\t", bt[i],

            "\t\t", wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f "%(total\_wt /n) )

    print("Average turn around time = %.5f "% (total\_tat / n))

# Driver code

if \_\_name\_\_ =="\_\_main\_\_":

    # Process id's

    proc = [1, 2, 3]

    n = 3

    # Burst time of all processes

    burst\_time = [10, 5, 8]

    # Time quantum

    quantum = 2;

    findavgTime(proc, n, burst\_time, quantum)

from collections import deque

def fifo\_page\_replacement(pages, capacity):

    page\_queue = deque(maxlen=capacity)

    page\_faults = 0

    for page in pages:

        if page not in page\_queue:

            print(f"Page {page} is loaded into the memory.")

            page\_queue.append(page)

            page\_faults += 1

        else:

            print(f"Page {page} is already in the memory.")

    print(f"\nTotal Page Faults: {page\_faults}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Example usage

    page\_references = [2, 3, 4, 2, 1, 3, 7, 5, 4, 3]

    memory\_capacity = 3

    fifo\_page\_replacement(page\_references, memory\_capacity)

def bestFit(blockSize, m, processSize, n):

    # Stores block id of the block

    # allocated to a process

    allocation = [-1] \* n

    # pick each process and find suitable

    # blocks according to its size ad

    # assign to it

    for i in range(n):

        # Find the best fit block for

        # current process

        bestIdx = -1

        for j in range(m):

            if blockSize[j] >= processSize[i]:

                if bestIdx == -1:

                    bestIdx = j

                elif blockSize[bestIdx] > blockSize[j]:

                    bestIdx = j

        # If we could find a block for

        # current process

        if bestIdx != -1:

            # allocate block j to p[i] process

            allocation[i] = bestIdx

            # Reduce available memory in this block.

            blockSize[bestIdx] -= processSize[i]

    print("Process No. Process Size  Block no.")

    for i in range(n):

        print(i + 1, "       ", processSize[i],

                                end = "      ")

        if allocation[i] != -1:

            print(allocation[i] + 1)

        else:

            print("Not Allocated")

#   blockSize = [100, 500, 200, 300, 600]

#   processSize = [212, 417, 112, 426]

if \_\_name\_\_ == '\_\_main\_\_':

    blockSize = list(map(int, input("Enter memory block sizes (comma-separated): ").split(',')))

    m = len(blockSize)

    processSize = list(map(int, input("Enter process sizes (comma-separated): ").split(',')))

    n = len(processSize)

    bestFit(blockSize, m, processSize, n)

def worstFit(blockSize, m, processSize, n):

    # Stores block id of the block

    # allocated to a process

    # Initially no block is assigned

    # to any process

    allocation = [-1] \* n

    # pick each process and find suitable blocks

    # according to its size ad assign to it

    for i in range(n):

        # Find the best fit block for

        # current process

        wstIdx = -1

        for j in range(m):

            if blockSize[j] >= processSize[i]:

                if wstIdx == -1:

                    wstIdx = j

                elif blockSize[wstIdx] < blockSize[j]:

                    wstIdx = j

        # If we could find a block for

        # current process

        if wstIdx != -1:

            # allocate block j to p[i] process

            allocation[i] = wstIdx

            # Reduce available memory in this block.

            blockSize[wstIdx] -= processSize[i]

    print("Process No. Process Size Block no.")

    for i in range(n):

        print(i + 1, "       ",

            processSize[i], end = "  ")

        if allocation[i] != -1:

            print(allocation[i] + 1)

        else:

            print("Not Allocated")

if \_\_name\_\_ == '\_\_main\_\_':

    blockSize = [100, 500, 200, 300, 600]

    processSize = [212, 417, 112, 426]

    m = len(blockSize)

    n = len(processSize)

    worstFit(blockSize, m, processSize, n)

from collections import OrderedDict

class LRUCache:

    def \_\_init\_\_(self, capacity):

        self.cache = OrderedDict()

        self.capacity = capacity

    def refer(self, page):

        if page in self.cache:

            # Move the page to the end to mark it as most recently used

            self.cache.move\_to\_end(page)

        else:

            # Check if the cache is full

            if len(self.cache) >= self.capacity:

                # Remove the least recently used page (the first item in the ordered dictionary)

                self.cache.popitem(last=False)

            # Add the new page to the cache

            self.cache[page] = None

def lru\_page\_replacement(pages, capacity):

    lru\_cache = LRUCache(capacity)

    page\_faults = 0

    for page in pages:

        if page not in lru\_cache.cache:

            print(f"Page {page} is loaded into the memory.")

            lru\_cache.refer(page)

            page\_faults += 1

        else:

            print(f"Page {page} is already in the memory.")

    print(f"\nTotal Page Faults: {page\_faults}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Example usage

    page\_references = [2, 3, 4, 2, 1, 3, 7, 5, 4, 3]

    memory\_capacity = 3

    lru\_page\_replacement(page\_references, memory\_capacity)

def optimal\_page\_replacement(pages, capacity):

    page\_faults = 0

    page\_frames = [-1] \* capacity

    for i in range(len(pages)):

        if pages[i] not in page\_frames:

            if -1 in page\_frames:

                # If there is an empty frame, place the page in it

                index = page\_frames.index(-1)

                page\_frames[index] = pages[i]

            else:

                # Find the page that will not be used for the longest period in the future

                future\_occurrences = {page: float('inf') for page in page\_frames}

                for j in range(i + 1, len(pages)):

                    if pages[j] in future\_occurrences:

                        future\_occurrences[pages[j]] = j

                page\_to\_replace = max(future\_occurrences, key=future\_occurrences.get)

                index = page\_frames.index(page\_to\_replace)

                page\_frames[index] = pages[i]

            print(f"Page {pages[i]} is loaded into the memory.")

            page\_faults += 1

        else:

            print(f"Page {pages[i]} is already in the memory.")

    print(f"\nTotal Page Faults: {page\_faults}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Example usage

    page\_references = [2, 3, 4, 2, 1, 3, 7, 5, 4, 3]

    memory\_capacity = 3

    optimal\_page\_replacement(page\_references, memory\_capacity)

def firstFit(blockSize, m, processSize, n):

    # Stores block id of the

    # block allocated to a process

    allocation = [-1] \* n

    # Initially no block is assigned to any process

    # pick each process and find suitable blocks

    # according to its size ad assign to it

    for i in range(n):

        for j in range(m):

            if blockSize[j] >= processSize[i]:

                # allocate block j to p[i] process

                allocation[i] = j

                # Reduce available memory in this block.

                blockSize[j] -= processSize[i]

                break

    print(" Process No. Process Size     Block no.")

    for i in range(n):

        print(" ", i + 1, "      ", processSize[i],

                        "        ", end = " ")

        if allocation[i] != -1:

            print(allocation[i] + 1)

        else:

            print("Not Allocated")

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    blockSize = [100, 500, 200, 300, 600]

    processSize = [212, 417, 112, 426]

    m = len(blockSize)

    n = len(processSize)

    firstFit(blockSize, m, processSize, n)

def NextFit(blockSize, m, processSize, n):

    allocation = [-1] \* n

    j = 0

    t = m-1

    # pick each process and find suitable blocks

    # according to its size ad assign to it

    for i in range(n):

        # Do not start from beginning

        while j < m:

            if blockSize[j] >= processSize[i]:

                # allocate block j to p[i] process

                allocation[i] = j

                # Reduce available memory in this block.

                blockSize[j] -= processSize[i]

                # sets a new end point

                t = (j - 1) % m

                break

            if t == j:

                # sets a new end point

                t = (j - 1) % m

                # breaks the loop after going through all memory block

                break

            # mod m will help in traversing the

            # blocks from starting block after

            # we reach the end.

            j = (j + 1) % m

    print("Process No. Process Size Block no.")

    for i in range(n):

        print("\t", i + 1, "\t", processSize[i],end = "\t\t")

        if allocation[i] != -1:

            print(allocation[i] + 1)

        else:

            print("Not Allocated")

# Driver Code

if \_\_name\_\_ == '\_\_main\_\_':

    blockSize = [100,500,200,300,600]

    processSize = [212,417,112,426]

    m = len(blockSize)

    n = len(processSize)

    NextFit(blockSize, m, processSize, n)

class SymbolTable:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.symbols = {}

    def add\_symbol(self, label, address):

        if label not in self.symbols:

            self.symbols[label] = address

    def get\_address(self, label):

        return self.symbols.get(label, None)

    def \_\_str\_\_(self):  # Fixed string representation method name

        return str(self.symbols)

class LiteralTable:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.literals = []

        self.current\_pool = 0

        self.pool\_table = []

    def add\_literal(self, literal):

        if literal not in (lit for lit, \_ in self.literals):

            self.literals.append((literal, None))

    def allocate\_literals(self, start\_address):

        address = start\_address

        for i, (literal, addr) in enumerate(self.literals):

            if addr is None:

                self.literals[i] = (literal, address)

                address += 1

        self.pool\_table.append(len(self.literals))

    def get\_address(self, literal):

        for lit, addr in self.literals:

            if lit == literal:

                return addr

        return None

    def \_\_str\_\_(self):  # Fixed string representation method name

        literals\_str = "\n".join(f"{literal}: {addr}" for literal, addr in self.literals)

        pool\_str = "\n".join(f"Pool {i+1}: {start}" for i, start in enumerate(self.pool\_table))

        return f"Literals:\n{literals\_str}\n\nPools:\n{pool\_str}"

class IntermediateCode:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.instructions = []

    def add\_instruction(self, address, instruction, operand):

        self.instructions.append((address, instruction, operand))

    def \_\_str\_\_(self):  # Fixed string representation method name

        return '\n'.join(f"{addr}: {instr} {oprnd}" for addr, instr, oprnd in self.instructions)

class AssemblerPass1:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.symbol\_table = SymbolTable()

        self.literal\_table = LiteralTable()

        self.intermediate\_code = IntermediateCode()

        self.location\_counter = 0

        self.directives = {

            'START': self.start\_directive,

            'END': self.end\_directive,

            'LTORG': self.ltor\_directive

        }

    def start\_directive(self, operand):

        self.location\_counter = int(operand)

    def end\_directive(self, operand):

        self.literal\_table.allocate\_literals(self.location\_counter)

    def ltor\_directive(self, operand):

        self.literal\_table.allocate\_literals(self.location\_counter)

    def process\_line(self, line):

        label, instruction, operand = None, None, None

        parts = line.split()

        if len(parts) == 3:

            label, instruction, operand = parts

        elif len(parts) == 2:

            instruction, operand = parts

        elif len(parts) == 1:

            instruction = parts[0]

        if label:

            self.symbol\_table.add\_symbol(label, self.location\_counter)

        if instruction in self.directives:

            self.directives[instruction](operand)

        else:

            if operand and operand.startswith('='):

                self.literal\_table.add\_literal(operand)

            self.intermediate\_code.add\_instruction(self.location\_counter, instruction, operand)

            self.location\_counter += 1

    def assemble(self, source\_code):

        for line in source\_code:

            self.process\_line(line)

        return self.symbol\_table, self.literal\_table, self.intermediate\_code

# Sample source code for a pseudo-machine

source\_code = [

    "START 100",

    "LOOP LOAD =3",

    "ADD =6",

    "STORE A",

    "LTORG",

    "B STORE =10",

    "END"

]

assembler = AssemblerPass1()

symbol\_table, literal\_table, intermediate\_code = assembler.assemble(source\_code)

print("Symbol Table:")

print(symbol\_table)

print("\nLiteral Table and Pools:")

print(literal\_table)

print("\nIntermediate Code:")

print(intermediate\_code)

class SymbolTable:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.symbols = {}

    def add\_symbol(self, label, address):

        self.symbols[label] = address

    def get\_address(self, label):

        return self.symbols.get(label)

    def \_\_str\_\_(self):  # Fixed string representation method name

        result = "Label       Address\n"

        result += "-" \* 20 + "\n"

        for label, address in self.symbols.items():

            result += f"{label:<10} {address:<10}\n"

        return result

class LiteralTable:

    def \_\_init\_\_(self):  # Fixed constructor name

        self.literals = []

        self.pool\_table = []

    def add\_literal(self, literal):

        if literal not in [lit for lit, \_ in self.literals]:

            self.literals.append((literal, None))  # Add literals with None as address initially

    def allocate\_literals(self, start\_address):

        address = start\_address

        for i, (literal, addr) in enumerate(self.literals):

            if addr is None:  # Only allocate address if not already allocated

                self.literals[i] = (literal, address)

                address += 1

        self.pool\_table.append(len(self.literals))

    def get\_address(self, literal):

        for lit, addr in self.literals:

            if lit == literal:

                return addr

        return None

    def \_\_str\_\_(self):  # Fixed string representation method name

        literals\_str = "Literal      Address\n"

        literals\_str += "-" \* 20 + "\n"

        for literal, addr in self.literals:

            literals\_str += f"{literal:<10} {addr:<10}\n"

        pool\_str = "Pool Table\n"

        pool\_str += "-" \* 20 + "\n"

        for i, start in enumerate(self.pool\_table):

            pool\_str += f"Pool {i+1}:   {start}\n"

        return f"{literals\_str}\n{pool\_str}"

class IntermediateCode:

    def \_\_init\_\_(self, instructions):  # Fixed constructor name

        self.instructions = instructions

    def \_\_str\_\_(self):  # Fixed string representation method name

        result = "Address   Instruction   Operand\n"

        result += "-" \* 30 + "\n"

        for addr, instr, operand in self.instructions:

            result += f"{addr:<9} {instr:<12} {operand if operand else '':<9}\n"

        return result

class AssemblerPass1:

    def \_\_init\_\_(self):

        self.symbol\_table = SymbolTable()

        self.literal\_table = LiteralTable()

        self.intermediate\_code = IntermediateCode([])

        self.location\_counter = 0

        self.directives = {

            'START': self.start\_directive,

            'END': self.end\_directive,

            'LTORG': self.ltor\_directive

        }

    def start\_directive(self, operand):

        self.location\_counter = int(operand)

    def end\_directive(self, operand):

        self.literal\_table.allocate\_literals(self.location\_counter)

    def ltor\_directive(self, operand):

        self.literal\_table.allocate\_literals(self.location\_counter)

    def process\_line(self, line):

        label, instruction, operand = None, None, None

        parts = line.split()

        if len(parts) == 3:

            label, instruction, operand = parts

        elif len(parts) == 2:

            instruction, operand = parts

        elif len(parts) == 1:

            instruction = parts[0]

        if label:

            self.symbol\_table.add\_symbol(label, self.location\_counter)

        if instruction in self.directives:

            self.directives[instruction](operand)

        else:

            if operand and operand.startswith('='):

                self.literal\_table.add\_literal(operand)

            self.intermediate\_code.instructions.append((self.location\_counter, instruction, operand))

            self.location\_counter += 1

    def assemble(self, source\_code):

        for line in source\_code:

            self.process\_line(line)

        return self.symbol\_table, self.literal\_table, self.intermediate\_code

# Sample source code for a pseudo-machine

source\_code = [

    "START 100",

    "LOOP LOAD =5",

    "ADD =10",

    "STORE A",

    "LTORG",

    "B STORE =15",

    "END"

]

assembler = AssemblerPass1()

symbol\_table, literal\_table, intermediate\_code = assembler.assemble(source\_code)

print("Symbol Table:")

print(symbol\_table)

print("\nLiteral Table and Pools:")

print(literal\_table)

print("\nIntermediate Code:")

print(intermediate\_code)

class Pass2Assembler:

    def \_\_init\_\_(self, symbol\_table, literal\_table, intermediate\_code):

        self.symbol\_table = symbol\_table

        self.literal\_table = literal\_table

        self.intermediate\_code = intermediate\_code

        self.opcode\_mapping = {

            "LOAD": "01",

            "ADD": "02",

            "STORE": "03",

            # Add more opcodes here as needed

        }

    def generate\_machine\_code(self):

        machine\_code = []

        print(f"{'Address':<10} {'Instruction':<15} {'Operand':<15} {'Machine Code':<15}")

        print("=" \* 60)

        for addr, instr, operand in self.intermediate\_code.instructions:

            code = self.opcode\_mapping.get(instr, "??")  # Fallback if opcode is not recognized

            if operand:

                if operand.startswith('='):

                    # Handling literals

                    literal\_address = self.literal\_table.get\_address(operand)

                    if literal\_address is not None:

                        code += f"{literal\_address:02d}"

                    else:

                        code += "??"  # Fallback if literal address is not recognized

                elif operand in self.symbol\_table.symbols:

                    code += f"{self.symbol\_table.get\_address(operand):02d}"

                else:

                    code += "??"  # Fallback if operand address is not recognized

            else:

                code += "00"  # Default operand if none provided

            machine\_code.append((addr, instr, operand, code))

        for addr, instr, operand, code in machine\_code:

            print(f"{addr:<10} {instr:<15} {operand:<15} {code:<15}")

# Sample symbol table generated from Pass 1

symbol\_table = SymbolTable()

symbol\_table.add\_symbol("LOOP", 100)

symbol\_table.add\_symbol("A", 103)

symbol\_table.add\_symbol("B", 105)

# Sample literal table generated from Pass 1

literal\_table = LiteralTable()

literal\_table.add\_literal("=5")

literal\_table.add\_literal("=10")

literal\_table.add\_literal("=15")

literal\_table.allocate\_literals(104)

# Sample intermediate code generated from Pass 1

instructions = [

    (100, "LOAD", "=5"),

    (101, "ADD", "=10"),

    (102, "STORE", "A"),

    (103, "STORE", "=15")

]

intermediate\_code = IntermediateCode(instructions)

# Perform Pass 2

pass2 = Pass2Assembler(symbol\_table, literal\_table, intermediate\_code)

pass2.generate\_machine\_code()

import re

# Step 1: Define a function to identify and store macros

def store\_macros(source\_code):

    macros = {}  # Dictionary to store macros

    # Regular expression to identify macro definitions

    macro\_pattern = re.compile(r"^(\w+)\s+MACRO(.\*?)(?=\s+ENDMACRO|$)", re.DOTALL)

    # Iterate through the source code and identify macro definitions

    lines = source\_code.split("\n")

    for line in lines:

        match = macro\_pattern.match(line.strip())

        if match:

            macro\_name = match.group(1)

            macro\_body = match.group(2).strip()

            macros[macro\_name] = macro\_body

    return macros

# Step 2: Replace macro invocations with the corresponding definitions

def replace\_macros\_with\_definitions(source\_code, macros):

    # Regular expression to identify macro invocations

    invocation\_pattern = re.compile(r"\b(\w+)\b")

    # Replace invocations with macro definitions

    lines = source\_code.split("\n")

    processed\_lines = []

    for line in lines:

        line = line.strip()

        if invocation\_pattern.match(line):

            # Check if the line is a macro invocation

            words = line.split()

            for i, word in enumerate(words):

                if word in macros:

                    # Replace macro invocation with macro body

                    words[i] = macros[word]

            processed\_lines.append(" ".join(words))

        else:

            processed\_lines.append(line)

    return "\n".join(processed\_lines)

# Main function to simulate the two-pass macro processor

def pass\_one\_of\_macro\_processor(source\_code):

    macros = store\_macros(source\_code)  # Store macros from the source code

    print("Stored Macros:")

    for macro\_name, macro\_body in macros.items():

        print(f"{macro\_name}: {macro\_body}")

    processed\_code = replace\_macros\_with\_definitions(source\_code, macros)  # Replace invocations

    return processed\_code

# Example Source Code with Macro Definitions and Invocations

source\_code = """

MACRO    ADD

    MOV AX, BX

    ADD AX, CX

ENDMACRO

MACRO    SUB

    MOV AX, BX

    SUB AX, CX

ENDMACRO

    ADD

    SUB

"""

# Call Pass-I

processed\_code = pass\_one\_of\_macro\_processor(source\_code)

# Print the processed code

print("\nProcessed Code (After Pass-I):")

print(processed\_code)

import re

# Pass-I: Store macros

def store\_macros(source\_code):

    macros = {}  # Dictionary to store macros

    # Regular expression to identify macro definitions

    macro\_pattern = re.compile(r"^(\w+)\s+MACRO(.\*?)(?=\s+ENDMACRO|$)", re.DOTALL)

    # Iterate through the source code and identify macro definitions

    lines = source\_code.split("\n")

    for line in lines:

        match = macro\_pattern.match(line.strip())

        if match:

            macro\_name = match.group(1)

            macro\_body = match.group(2).strip()

            macros[macro\_name] = macro\_body

    return macros

# Pass-I: Replace macros with definitions

def replace\_macros\_with\_definitions(source\_code, macros):

    # Regular expression to identify macro invocations

    invocation\_pattern = re.compile(r"\b(\w+)\b")

    # Replace invocations with macro definitions

    lines = source\_code.split("\n")

    processed\_lines = []

    for line in lines:

        line = line.strip()

        if invocation\_pattern.match(line):

            # Check if the line is a macro invocation

            words = line.split()

            for i, word in enumerate(words):

                if word in macros:

                    # Replace macro invocation with macro body

                    words[i] = macros[word]

            processed\_lines.append(" ".join(words))

        else:

            processed\_lines.append(line)

    return "\n".join(processed\_lines)

# Pass-II: Generate the final code (machine or intermediate code)

def generate\_final\_code(expanded\_code):

    # For simplicity, assume we convert the expanded code to an intermediate code

    # For now, we are just returning the expanded code as is.

    final\_code = []

    lines = expanded\_code.split("\n")

    for line in lines:

        line = line.strip()

        if line:  # Only process non-empty lines

            # For each line, generate intermediate code (just for demonstration)

            final\_code.append(f"Generated: {line}")

    return "\n".join(final\_code)

# Main function to simulate the entire macro processor

def macro\_processor(source\_code):

    macros = store\_macros(source\_code)  # Pass-I: Store macros

    print("Stored Macros:")

    for macro\_name, macro\_body in macros.items():

        print(f"{macro\_name}: {macro\_body}")

    expanded\_code = replace\_macros\_with\_definitions(source\_code, macros)  # Pass-I: Replace invocations

    print("\nExpanded Code After Pass-I:")

    print(expanded\_code)

    final\_code = generate\_final\_code(expanded\_code)  # Pass-II: Generate final code

    print("\nFinal Code After Pass-II:")

    print(final\_code)

# Example Source Code with Macro Definitions and Invocations

source\_code = """

MACRO    ADD

    MOV AX, BX

    ADD AX, CX

ENDMACRO

MACRO    SUB

    MOV AX, BX

    SUB AX, CX

ENDMACRO

    ADD

    SUB

"""

# Run the Macro Processor (Pass-I + Pass-II)

macro\_processor(source\_code)

def findWaitingTime(processes, n, bt, wt):

    wt[0] = 0  # Waiting time for the first process is 0

    for i in range(1, n):

        wt[i] = bt[i - 1] + wt[i - 1]  # Calculate waiting time for each process

def findTurnAroundTime(processes, n, bt, wt, tat):

    for i in range(n):

        tat[i] = bt[i] + wt[i]  # Turnaround time = Burst time + Waiting time

def findavgTime(processes, n, bt):

    wt = [0] \* n  # Waiting time array

    tat = [0] \* n  # Turnaround time array

    total\_wt = 0  # Total waiting time

    total\_tat = 0  # Total turnaround time

    findWaitingTime(processes, n, bt, wt)  # Calculate waiting time

    findTurnAroundTime(processes, n, bt, wt, tat)  # Calculate turnaround time

    print("Processes Burst time Waiting time Turn around time")

    for i in range(n):

        total\_wt += wt[i]

        total\_tat += tat[i]

        print(f"{processes[i]}\t\t{bt[i]}\t {wt[i]}\t\t {tat[i]}")

    print(f"Average waiting time = {total\_wt / n}")

    print(f"Average turn around time = {total\_tat / n}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Process ids

    processes = [1, 2, 3]

    n = len(processes)

    # Burst time of all processes

    burst\_time = [10, 5, 8]

    findavgTime(processes, n, burst\_time)

void setup() {

// put your setup code here, to run once:

pinMode(A0, INPUT); //IR sensor to CN11

pinMode(0, OUTPUT);

pinMode(1, OUTPUT);

pinMode(2, OUTPUT);

pinMode(4, OUTPUT);

pinMode(13, OUTPUT);

}

void loop() {

// put your main code here, to run repeatedly:

if(digitalRead(A0) == 1)

{

digitalWrite(0,HIGH);

digitalWrite(1,HIGH);

digitalWrite(2,HIGH);

digitalWrite(4,HIGH);

digitalWrite(13,HIGH);

} else {

digitalWrite(0,LOW);

digitalWrite(1, LOW);

digitalWrite(2, LOW);

digitalWrite(4, LOW);

digitalWrite(13, LOW);

}

}

#include <SD.h>

#include "DHT.h"

#define DHTPIN 4

#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

void setup() {

Serial.begin(9600);

Serial.println("Humidity AND Temperature");

dht.begin();

}

void loop() {

delay(2000);

float h = dht.readHumidity();

float t = dht.readTemperature();

Serial.print("\nHumidity:");

Serial.print(h);

Serial.print("\n% Temperature :");

Serial.print(t);

Serial.print("\*C");

}

# math\_operations.py

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

def multiply(a, b):

    return a \* b

def divide(a, b):

    if b == 0:

        raise ValueError("Division by zero is not allowed.")

    return a / b

if \_\_name\_\_ == "\_\_main\_\_":

    a, b = 10, 5

    print("Addition:", add(a, b))

    print("Subtraction:", subtract(a, b))

    print("Multiplication:", multiply(a, b))

    print("Division:", divide(a, b))

n = int(input("Enter the number of processes: "))

p = [0] \* 10

pp = [0] \* 10

bt = [0] \* 10

w = [0] \* 10

t = [0] \* 10

print("Enter the burst time and priority for each process:")

for i in range(n):

    print(f"Process[{i + 1}]")

    bt[i] = int(input("Burst Time: "))

    pp[i] = int(input("Priority: "))

    p[i] = i + 1

for i in range(n - 1):

    for j in range(i + 1, n):

        if pp[i] < pp[j]:

            pp[i], pp[j] = pp[j], pp[i]

            bt[i], bt[j] = bt[j], bt[i]

            p[i], p[j] = p[j], p[i]

w[0] = 0

awt = 0

t[0] = bt[0]

atat = t[0]

for i in range(1, n):

    w[i] = t[i - 1]

    awt += w[i]

    t[i] = w[i] + bt[i]

    atat += t[i]

print("Process \t Burst time \t Wait time \t TAT \t Priority ")

for i in range(n):

    print(f"{p[i]}\t\t{bt[i]}\t\t{w[i]}\t\t{t[i]}\t\t{pp[i]}")

awt /= n

atat /= n

print(f"Average Wait time: {awt}")

print(f"Average TAT: {atat}")