Os  
  
shell programming:

1)arithmetic:  
  
#!/bin/bash

echo "addition of two number"

var=$((2+7))

echo $var

echo "substraction of two number"

var1=$((7-2))

echo $var1

echo "multiplication of two numbers"

var2=$((7\*2))

echo $var2

echo "divisionof 2 numbers:"

var3=$((4/4))

echo $var3  
  
  
  
2)substring:  
#!/bin/bash

# Read inputs

read -p "Enter the main string: " main\_string

read -p "Enter the substring to search for: " search\_substring

# Find the position of the substring

position=$(expr match "$main\_string" ".\*$search\_substring")

# Check if the substring exists and output result

if [[ $position -gt 0 ]]; then

start\_position=$((position - ${#search\_substring} + 1))

echo "The substring '$search\_substring' is found at position: $start\_position"

else

echo "The substring '$search\_substring' is not found in the main string."

fi  
  
  
3)sorting :bubblesort  
#!/bin/bash

read -p "Enter integers separated by spaces: " -a arr

n=${#arr[@]}

echo $n

for ((i = 0; i<n; i++))

do

for ((j = 0; j<n-1; j++))

do

if [ ${arr[j]} -gt ${arr[$((j+1))]} ]

then

temp=${arr[j]}

arr[$j]=${arr[$((j+1))]}

arr[$((j+1))]=$temp

fi

done

done

echo "Array in sorted order :"

echo ${arr[@]}

insertion :  
#insertion sort shell

#!/bin/bash

echo "Enter the number of elements:"

read n

echo "Enter the elements:"

for ((i = 0; i < n; i++))

do

read arr[i]

done

for ((i = 1; i < n; i++))

do

key=${arr[i]}

j=$((i - 1))

while ((j >= 0 && arr[j] > key))

do

arr[j + 1]=${arr[j]}

j=$((j - 1))

done

arr[j + 1]=$key

done

echo "Sorted array using Insertion Sort:"

echo "${arr[@]}"

selection sort  
#selection sort shell

#!/bin/bash

echo "Enter the number of elements:"

read n

echo "Enter the elements:"

for ((i = 0; i < n; i++))

do

read arr[i]

done

for ((i = 0; i < n - 1; i++))

do

min\_index=$i

for ((j = i + 1; j < n; j++))

do

if ((arr[j] < arr[min\_index]))

then

min\_index=$j

fi

done

temp=${arr[i]}

arr[i]=${arr[min\_index]}

arr[min\_index]=$temp

done

echo "Sorted array using Selection Sort:"

echo "${arr[@]}"

4)upper to lower:  
  
#!/bin/bash

read -p "enter the string in uppercase to convert to lowercase" utl

t=${utl,,}

echo "$t"

read -p "enter the string in lower to convert to uppercase" ltu

t1=${ltu^^}

echo "$t1"

5)palindrome:  
#!/bin/bash

read -p "enter a string " input\_string

reversed\_string=$(echo "$input\_string" |rev)

if [[ "$input\_string" == "reversed\_string" ]]; then

echo "the entered string '$input\_string' is a pallindrome ."

else

echo "the entered string '$input\_string' is not a pallindrome."

Fi

Fibopnaccii and extra

#factorial shell

#!/usr/bin/bash

factorial() {

local num=$1

if [ "$num" -le 1 ]; then

echo 1

else

echo $((num \* $(factorial $((num - 1)))))

fi

}

factorial 5

#fibonacci shell

#!/usr/bin/bash

fibonacci() {

local num=$1

if [ "$num" -le 0 ]; then

echo 0

elif [ "$num" -eq 1 ]; then

echo 1

else

echo $(( $(fibonacci $((num - 1))) + $(fibonacci $((num - 2))) ))

fi

}

for i in {0..9}; do

fibonacci $i

done

#Replace 20% line of a file with patterns shell

#!/bin/bash

file="$1"

pattern="CHINTU"

total\_lines=$(wc -l < "$file")

lines\_to\_replace=$((total\_lines / 5))

for ((i = 1; i <= lines\_to\_replace; i++)); do

line=$(( (RANDOM % total\_lines) + 1 ))

sed -i "${line}s|.\*|$pattern|" "$file"

done

awk:  
  
1)student :  
student.awk  
  
BEGIN{

FS=" : ";

OFS ="\t";

print "student report"

print "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"

print "Name \t Average \t Grade"

}

NR >1 {

avg=($3 +$4 +$5) /3

if (avg <=50)

grade ="\t F"

else if (avg > 50)

grade ="\t Pass"

print $2 ,avg ,grade

}

END{

print "end of report"

print " \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"

}

input.txt:  
  
id: name : m1 : m2 : m3

1 : a : 50 : 70 : 60

2 : b : 60 : 90 : 86

3 : c : 63 : 50 : 77  
  
  
2) employee  
  
employee.awk  
BEGIN {

print "Report"

}

{

sal = $3

da = 0.50 \* sal

hra = 0.30 \* sal

gs = sal - da - hra

grade = "F"

print $2, da, hra, gs

}

END {

print "END"

}

input.txt  
  
1 a 10000 20 35

2 B 32222 12 45

3 c 32224 45 30  
  
  
  
or  
  
  
#Awk program to calculate grade of Student

BEGIN{

print "Report"

}

{

avg=($3+$4+$5)/3

grade = "F"

if (avg >= 90) {

grade = "A+"

} else if (avg >= 80) {

grade = "A"

} else if (avg >= 70) {

grade = "B"

} else if (avg >= 60) {

grade = "C"

} else if (avg >= 50) {

grade = "D"

} else if (avg >= 40) {

grade = "E"

}

print $2, grade

}

END{

print "END"

}

# input for the student awk program:

1 a 12 20 35

2 B 32 12 45

3 c 34 45 30

#Awk program to calculate gross salary of employee

BEGIN{

print "Report"

}

{

sal=$3

da=0.50\*sal

hra=0.30\*sal

gs=sal-da-hra

print $2,da,hra,gs

}

END{

print "END"

}

#awk prog input:

1 a 10000

2 B 7000

3 c 20000

assign

producer consumer  
mutex

import threading

import random

import time

BUFFER\_SIZE = 5

buffer = [None] \* BUFFER\_SIZE

in\_pos = out\_pos = count = 0

# Mutex lock and condition variables

buffer\_lock = threading.Lock()

buffer\_not\_empty = threading.Condition(buffer\_lock)

buffer\_not\_full = threading.Condition(buffer\_lock)

def producer(producer\_id):

    global in\_pos, count

    for \_ in range(10):

        item = random.randint(1, 100)

        with buffer\_not\_full:

            while count == BUFFER\_SIZE:

                print(f"Producer {producer\_id} blocked, buffer is full.")

                buffer\_not\_full.wait()

            buffer[in\_pos] = item

            print(f"Producer {producer\_id} inserted {item} in slot {in\_pos}.")

            in\_pos = (in\_pos + 1) % BUFFER\_SIZE

            count += 1

            buffer\_not\_empty.notify()

        time.sleep(random.uniform(0.1, 0.5))

def consumer(consumer\_id):

    global out\_pos, count

    for \_ in range(10):

        with buffer\_not\_empty:

            while count == 0:

                print(f"Consumer {consumer\_id} blocked, buffer is empty.")

                buffer\_not\_empty.wait()

            item = buffer[out\_pos]

            print(f"Consumer {consumer\_id} consumed {item} from slot {out\_pos}.")

            out\_pos = (out\_pos + 1) % BUFFER\_SIZE

            count -= 1

            buffer\_not\_full.notify()

        time.sleep(random.uniform(0.1, 0.5))

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

    # Start 2 producers and 2 consumers

    threads = [threading.Thread(target=producer, args=(i+1,)) for i in range(2)] + \

              [threading.Thread(target=consumer, args=(i+1,)) for i in range(2)]

    for t in threads:

        t.start()

    for t in threads:

        t.join()

    print("Producer-Consumer process finished.")

producers semaphore  
import threading

import random

import time

BUFFER\_SIZE = 5

buffer = [None] \* BUFFER\_SIZE

in\_pos = out\_pos = count = 0

# Semaphore objects

empty\_slots = threading.Semaphore(BUFFER\_SIZE)  # Tracks empty slots in the buffer

filled\_slots = threading.Semaphore(0)           # Tracks filled slots in the buffer

buffer\_lock = threading.Semaphore(1)            # Mutex for accessing the buffer

def producer(producer\_id):

    global in\_pos, count

    for \_ in range(10):

        item = random.randint(1, 100)

        empty\_slots.acquire()  # Wait for an empty slot

        with buffer\_lock:      # Lock the buffer for insertion

            buffer[in\_pos] = item

            print(f"Producer {producer\_id} inserted {item} in slot {in\_pos}.")

            in\_pos = (in\_pos + 1) % BUFFER\_SIZE

            count += 1

        filled\_slots.release()  # Signal that a slot is filled

        time.sleep(random.uniform(0.1, 0.5))

def consumer(consumer\_id):

    global out\_pos, count

    for \_ in range(10):

        filled\_slots.acquire()  # Wait for a filled slot

        with buffer\_lock:       # Lock the buffer for removal

            item = buffer[out\_pos]

            print(f"Consumer {consumer\_id} consumed {item} from slot {out\_pos}.")

            out\_pos = (out\_pos + 1) % BUFFER\_SIZE

            count -= 1

        empty\_slots.release()   # Signal that a slot is empty

        time.sleep(random.uniform(0.1, 0.5))

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

    # Start 2 producers and 2 consumers

    threads = [threading.Thread(target=producer, args=(i+1,)) for i in range(2)] + \

              [threading.Thread(target=consumer, args=(i+1,)) for i in range(2)]

    for t in threads:

        t.start()

    for t in threads:

        t.join()

    print("Producer-Consumer process finished.")

readers mutex  
import threading

import time

# Locks (Mutex)

resource\_lock = threading.Lock()

read\_count\_lock = threading.Lock()

read\_count = 0

def reader(reader\_id):

    global read\_count

    read\_count\_lock.acquire()

    read\_count += 1

    if read\_count == 1:

        resource\_lock.acquire()  # First reader locks the resource

    read\_count\_lock.release()

    # Reading

    print(f"Reader {reader\_id} is reading.")

    time.sleep(1)

    print(f"Reader {reader\_id} finished reading at {time.strftime('%H:%M:%S')}")

    read\_count\_lock.acquire()

    read\_count -= 1

    if read\_count == 0:

        resource\_lock.release()  # Last reader unlocks the resource

    read\_count\_lock.release()

def writer(writer\_id):

    resource\_lock.acquire()

    # Writing

    print(f"Writer {writer\_id} is writing.")

    time.sleep(1)

    print(f"Writer {writer\_id} finished writing at {time.strftime('%H:%M:%S')}")

    resource\_lock.release()

# Create threads

readers = [threading.Thread(target=reader, args=(i+1,)) for i in range(3)]

writers = [threading.Thread(target=writer, args=(i+1,)) for i in range(2)]

# Start threads

for t in readers + writers:

    t.start()

# Wait for threads to finish

for t in readers + writers:

    t.join()

readers semaphore:  
import threading

import time

# Semaphores

resource = threading.Semaphore(1)

read\_count\_access = threading.Semaphore(1)

read\_count = 0

def reader(reader\_id):

    global read\_count

    read\_count\_access.acquire()

    read\_count += 1

    if read\_count == 1:

        resource.acquire()  # First reader locks the resource

    read\_count\_access.release()

    # Reading

    print(f"Reader {reader\_id} is reading.")

    time.sleep(1)

    print(f"Reader {reader\_id} finished reading at {time.strftime('%H:%M:%S')}")

    read\_count\_access.acquire()

    read\_count -= 1

    if read\_count == 0:

        resource.release()  # Last reader unlocks the resource

    read\_count\_access.release()

def writer(writer\_id):

    resource.acquire()

    # Writing

    print(f"Writer {writer\_id} is writing.")

    time.sleep(1)

    print(f"Writer {writer\_id} finished writing at {time.strftime('%H:%M:%S')}")

    resource.release()

# Create threads

readers = [threading.Thread(target=reader, args=(i+1,)) for i in range(3)]

writers = [threading.Thread(target=writer, args=(i+1,)) for i in range(2)]

# Start threads

for t in readers + writers:

    t.start()

# Wait for threads to finish

for t in readers + writers:

    t.join()

page replacement:  
  
class PageReplacement:

    def \_\_init\_\_(self, capacity, pages):

        self.capacity = capacity

        self.pages = pages

    def fifo(self):

        page\_frame = []

        page\_faults = 0

        for page in self.pages:

            if page not in page\_frame:

                if len(page\_frame) >= self.capacity:

                    page\_frame.pop(0)  # Remove the first page (FIFO)

                page\_frame.append(page)

                page\_faults += 1

        return page\_faults

    def lru(self):

        page\_frame = []

        page\_faults = 0

        time\_stamp = {}

        for time, page in enumerate(self.pages):

            if page not in page\_frame:

                if len(page\_frame) >= self.capacity:

                    lru\_page = min(time\_stamp, key=time\_stamp.get)  # Find the least recently used

                    page\_frame.remove(lru\_page)

                    del time\_stamp[lru\_page]

                page\_frame.append(page)

                time\_stamp[page] = time  # Update the timestamp

                page\_faults += 1

            else:

                time\_stamp[page] = time  # Update timestamp for existing pages

        return page\_faults

    def opt(self):

        page\_frame = []

        page\_faults = 0

        for i, page in enumerate(self.pages):

            if page not in page\_frame:

                if len(page\_frame) >= self.capacity:

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

                    future\_use = {}

                    for p in page\_frame:

                        future\_use[p] = float('inf')  # Default to infinity

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

                        if self.pages[j] in page\_frame and self.pages[j] not in future\_use:

                            future\_use[self.pages[j]] = j  # Track the future index

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

                    page\_frame.remove(page\_to\_replace)

                page\_frame.append(page)

                page\_faults += 1

        return page\_faults

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

    page\_replacement = PageReplacement(capacity, pages)

    fifo\_faults = page\_replacement.fifo()

    lru\_faults = page\_replacement.lru()

    opt\_faults = page\_replacement.opt()

    total\_requests = len(pages)

    total\_faults = fifo\_faults + lru\_faults + opt\_faults

    hit\_ratio\_fifo = (total\_requests - fifo\_faults) / total\_requests

    miss\_ratio\_fifo = fifo\_faults / total\_requests

    hit\_ratio\_lru = (total\_requests - lru\_faults) / total\_requests

    miss\_ratio\_lru = lru\_faults / total\_requests

    hit\_ratio\_opt = (total\_requests - opt\_faults) / total\_requests

    miss\_ratio\_opt = opt\_faults / total\_requests

    print("Page Replacement Simulation Results:")

    print(f"FIFO Page Faults: {fifo\_faults}, Hit Ratio: {hit\_ratio\_fifo:.2f}, Miss Ratio: {miss\_ratio\_fifo:.2f}")

    print(f"LRU Page Faults: {lru\_faults}, Hit Ratio: {hit\_ratio\_lru:.2f}, Miss Ratio: {miss\_ratio\_lru:.2f}")

    print(f"OPT Page Faults: {opt\_faults}, Hit Ratio: {hit\_ratio\_opt:.2f}, Miss Ratio: {miss\_ratio\_opt:.2f}")

# Example usage

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

    capacity = 3  # Number of frames

    pages = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0]  # Page reference string

    simulate\_page\_replacement(capacity, pages)

Simulate the following CPU scheduling algorithms:

1. First come First serve

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

def plot\_gantt\_chart(processes, start\_times, finish\_times):

    fig, gnt = plt.subplots()

    # Setting labels for x-axis and y-axis

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    # Setting ticks on y-axis

    gnt.set\_yticks([10 \* (i + 1) for i in range(len(processes))])

    gnt.set\_yticklabels([f'P{pid}' for pid, \_, \_ in processes])

    # Setting graph limits

    gnt.set\_xlim(0, max(finish\_times) + 5)

    gnt.set\_ylim(0, 10 \* (len(processes) + 1))

    # Plotting the bars (tasks)

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    for i, (pid, arrival, burst) in enumerate(processes):

        gnt.broken\_barh([(start\_times[i], burst)], (10 \* (i + 1) - 5, 9), facecolors=(colors[i % len(colors)]))

    # Adding legend

    handles = [mpatches.Patch(color=colors[i % len(colors)], label=f'P{pid}') for i, (pid, \_, \_) in enumerate(processes)]

    plt.legend(handles=handles)

    # Show the plot at the end

    plt.show()

def fcfs\_scheduling\_with\_gantt(processes):

    n = len(processes)

    processes.sort(key=lambda x: x[1])  # Sort by arrival time

    start\_time = 0

    start\_times, finish\_times = [], []

    turnaround\_times, waiting\_times = [], []

    for i, (pid, arrival, burst) in enumerate(processes):

        if start\_time < arrival:

            start\_time = arrival

        start\_times.append(start\_time)

        finish\_time = start\_time + burst

        finish\_times.append(finish\_time)

        # Turnaround time = Finish time - Arrival time

        turnaround\_time = finish\_time - arrival

        turnaround\_times.append(turnaround\_time)

        # Waiting time = Turnaround time - Burst time

        waiting\_time = turnaround\_time - burst

        waiting\_times.append(waiting\_time)

        start\_time = finish\_time

    # Print details before showing the Gantt chart

    print("\nProcess\tArrival\tBurst\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    # Call the Gantt chart plotting function

    plot\_gantt\_chart(processes, start\_times, finish\_times)

#  input: (Process ID, Arrival Time, Burst Time)

processes = [(1, 0, 4), (2, 1, 3), (3, 2, 1), (4, 3, 2)]

fcfs\_scheduling\_with\_gantt(processes)

    b. Shortest Job First (Non-preemptive)    
  
 import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

def plot\_gantt\_chart(processes, start\_times, finish\_times):

    fig, gnt = plt.subplots()

    # Setting labels for x-axis and y-axis

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    # Setting ticks on y-axis

    gnt.set\_yticks([10 \* (i + 1) for i in range(len(processes))])

    gnt.set\_yticklabels([f'P{pid}' for pid, \_, \_ in processes])

    # Plotting the bars (tasks)

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    for i, (pid, arrival, burst) in enumerate(processes):

        gnt.broken\_barh([(start\_times[i], burst)], (10 \* (i + 1) - 5, 9), facecolors=(colors[i % len(colors)]))

    plt.show()

def sjf\_non\_preemptive(processes):

    n = len(processes)

    processes.sort(key=lambda x: x[1])  # Sort by arrival time

    start\_time = 0

    completed = [False] \* n

    start\_times, finish\_times = [0] \* n, [0] \* n

    turnaround\_times, waiting\_times = [0] \* n, [0] \* n

    gantt\_chart = []

    for \_ in range(n):

        # Find the shortest process that has arrived but not completed

        idx = -1

        min\_burst = float('inf')

        for i, (pid, arrival, burst) in enumerate(processes):

            if arrival <= start\_time and not completed[i] and burst < min\_burst:

                min\_burst = burst

                idx = i

        if idx == -1:

            start\_time += 1  # If no process is ready, increment the start time

            continue

        start\_times[idx] = start\_time

        finish\_times[idx] = start\_time + processes[idx][2]  # finish time = start time + burst

        turnaround\_times[idx] = finish\_times[idx] - processes[idx][1]  # Turnaround = finish - arrival

        waiting\_times[idx] = turnaround\_times[idx] - processes[idx][2]  # Waiting = turnaround - burst

        completed[idx] = True

        start\_time = finish\_times[idx]

        gantt\_chart.append(f"P{processes[idx][0]}")

    # Print table

    print("\nProcess\tArrival\tBurst\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    # Call Gantt chart plotting

    plot\_gantt\_chart(processes, start\_times, finish\_times)

    print("\nGantt Chart Sequence:")

    print(" -> ".join(gantt\_chart))

# Example Input

processes = [(1, 0, 4), (2, 1, 2), (3, 2, 3), (4, 3, 5)]

sjf\_non\_preemptive(processes)

 c. Shortest Job First (Preemptive)  
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

# Function to plot the Gantt chart for SJF Preemptive

def plot\_gantt\_chart\_preemptive(gantt\_chart):

    fig, gnt = plt.subplots()

    # Setting labels for x-axis and y-axis

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    # Extract unique processes

    processes = sorted(set(pid for \_, pid in gantt\_chart if pid is not None))

    process\_indices = {pid: idx + 1 for idx, pid in enumerate(processes)}

    # Setting ticks on y-axis

    gnt.set\_yticks([10 \* process\_indices[pid] for pid in processes])

    gnt.set\_yticklabels([f'P{pid}' for pid in processes])

    # Setting graph limits

    gnt.set\_xlim(0, max(time for time, \_ in gantt\_chart) + 1)

    gnt.set\_ylim(0, 10 \* (len(processes) + 1))

    # Colors for processes

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    # Plot bars for each process in gantt\_chart

    for i, (time, pid) in enumerate(gantt\_chart):

        if pid is not None:

            gnt.broken\_barh([(time, 1)], (10 \* process\_indices[pid] - 5, 9),

                            facecolors=(colors[process\_indices[pid] % len(colors)]))

    # Adding legend

    handles = [mpatches.Patch(color=colors[process\_indices[pid] % len(colors)], label=f'P{pid}') for pid in processes]

    plt.legend(handles=handles)

    plt.show()

# Function to implement SJF Preemptive Scheduling

def sjf\_preemptive(processes):

    n = len(processes)

    remaining\_burst = [burst for \_, \_, burst in processes]

    start\_time = 0

    completed = 0

    current\_time = 0

    turnaround\_times, waiting\_times = [0] \* n, [0] \* n

    finish\_times = [-1] \* n

    start\_times = [-1] \* n

    gantt\_chart = []

    while completed < n:

        idx = -1

        min\_burst = float('inf')

        for i in range(n):

            pid, arrival, burst = processes[i]

            if arrival <= current\_time and remaining\_burst[i] < min\_burst and remaining\_burst[i] > 0:

                min\_burst = remaining\_burst[i]

                idx = i

        if idx == -1:

            gantt\_chart.append((current\_time, None))  # No process running

            current\_time += 1

            continue

        # Mark the start time if the process is first picked

        if start\_times[idx] == -1:

            start\_times[idx] = current\_time

        gantt\_chart.append((current\_time, processes[idx][0]))  # Log the process ID in Gantt chart

        remaining\_burst[idx] -= 1

        current\_time += 1

        if remaining\_burst[idx] == 0:

            completed += 1

            finish\_times[idx] = current\_time

            turnaround\_times[idx] = finish\_times[idx] - processes[idx][1]  # Turnaround time = finish - arrival

            waiting\_times[idx] = turnaround\_times[idx] - processes[idx][2]  # Waiting time = turnaround - burst

    # Print details

    print("\nProcess\tArrival\tBurst\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    # Call Gantt chart plotting

    plot\_gantt\_chart\_preemptive(gantt\_chart)

# Example Input

processes = [(1, 0, 6), (2, 1, 4), (3, 2, 2), (4, 3, 1)]

sjf\_preemptive(processes)

d. Round Robin      
 import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

def plot\_gantt\_chart(processes, timeline):

    fig, gnt = plt.subplots()

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    # Get all unique processes from the timeline

    unique\_processes = sorted(set(p for p, \_ in timeline))

    # Set y-ticks for each process

    gnt.set\_yticks([10 \* (i + 1) for i in range(len(unique\_processes))])

    gnt.set\_yticklabels([f'P{pid}' for pid in unique\_processes])

    gnt.set\_xlim(0, max(t for \_, t in timeline))

    gnt.set\_ylim(0, 10 \* (len(unique\_processes) + 1))

    # Plot timeline for each process

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    for pid, start in timeline:

        gnt.broken\_barh([(start, 1)], (10 \* (pid) - 5, 9), facecolors=(colors[(pid - 1) % len(colors)]))

    plt.show()

def round\_robin(processes, quantum):

    n = len(processes)

    remaining\_burst = [burst for \_, \_, burst in processes]

    current\_time = 0

    start\_times, finish\_times = [0] \* n, [0] \* n

    turnaround\_times, waiting\_times = [0] \* n, [0] \* n

    queue = []

    timeline = []  # For Gantt chart

    for i, (pid, arrival, burst) in enumerate(processes):

        queue.append(i) if arrival == 0 else None

    index = 0

    while queue:

        i = queue.pop(0)

        pid, arrival, burst = processes[i]

        if remaining\_burst[i] > quantum:

            current\_time += quantum

            remaining\_burst[i] -= quantum

            timeline.append((pid, current\_time - quantum))

            queue.append(i)  # Re-add to queue if not finished

        else:

            current\_time += remaining\_burst[i]

            finish\_times[i] = current\_time

            timeline.append((pid, current\_time - remaining\_burst[i]))

            turnaround\_times[i] = finish\_times[i] - arrival

            waiting\_times[i] = turnaround\_times[i] - burst

            remaining\_burst[i] = 0

        # Add processes that have arrived by now

        for j, (p, a, \_) in enumerate(processes):

            if a <= current\_time and j not in queue and remaining\_burst[j] > 0:

                queue.append(j)

    # Print table

    print("\nProcess\tArrival\tBurst\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    plot\_gantt\_chart(processes, timeline)

# Example input

processes = [(1, 0, 5), (2, 1, 3), (3, 2, 8), (4, 3, 6)]

time\_quantum = 2

round\_robin(processes, time\_quantum)

  e. Priority (Non-preemptive)

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

def plot\_gantt\_chart(processes, start\_times, finish\_times):

    fig, gnt = plt.subplots()

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    gnt.set\_yticks([10 \* (i + 1) for i in range(len(processes))])

    gnt.set\_yticklabels([f'P{pid}' for pid, \_, \_, \_ in processes])

    gnt.set\_xlim(0, max(finish\_times) + 5)

    gnt.set\_ylim(0, 10 \* (len(processes) + 1))

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    for i, (pid, arrival, burst, priority) in enumerate(processes):

        gnt.broken\_barh([(start\_times[i], burst)], (10 \* (i + 1) - 5, 9), facecolors=(colors[i % len(colors)]))

    handles = [mpatches.Patch(color=colors[i % len(colors)], label=f'P{pid}') for i, (pid, \_, \_, \_) in enumerate(processes)]

    plt.legend(handles=handles)

    plt.show()

def priority\_non\_preemptive(processes):

    n = len(processes)

    processes.sort(key=lambda x: (x[1], x[3]))  # Sort by arrival and priority

    start\_time = 0

    completed = [False] \* n

    start\_times, finish\_times = [0] \* n, [0] \* n

    turnaround\_times, waiting\_times = [0] \* n, [0] \* n

    for \_ in range(n):

        idx = -1

        highest\_priority = float('inf')

        for i, (pid, arrival, burst, priority) in enumerate(processes):

            if arrival <= start\_time and not completed[i] and priority < highest\_priority:

                highest\_priority = priority

                idx = i

        if idx == -1:

            start\_time += 1  # If no process is ready, increment the start time

            continue

        start\_times[idx] = start\_time

        finish\_times[idx] = start\_time + processes[idx][2]

        turnaround\_times[idx] = finish\_times[idx] - processes[idx][1]

        waiting\_times[idx] = turnaround\_times[idx] - processes[idx][2]

        completed[idx] = True

        start\_time = finish\_times[idx]

    # Print table

    print("\nProcess\tArrival\tBurst\tPriority\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst, priority) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{priority}\t\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    plot\_gantt\_chart(processes, start\_times, finish\_times)

# Example Input

processes = [(1, 0, 4, 2), (2, 1, 3, 1), (3, 2, 1, 4), (4, 3, 2, 3)]

priority\_non\_preemptive(processes)

 f. Priority (preemptive)

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

def plot\_gantt\_chart(processes, timeline):

    fig, gnt = plt.subplots()

    gnt.set\_xlabel('Time')

    gnt.set\_ylabel('Processes')

    # Get all unique processes from the timeline

    unique\_processes = sorted(set(p for p, \_ in timeline))

    gnt.set\_yticks([10 \* (i + 1) for i in range(len(unique\_processes))])

    gnt.set\_yticklabels([f'P{pid}' for pid in unique\_processes])

    gnt.set\_xlim(0, max(t for \_, t in timeline) + 1)

    gnt.set\_ylim(0, 10 \* (len(unique\_processes) + 1))

    colors = ['red', 'green', 'blue', 'orange', 'purple']

    for pid, start in timeline:

        gnt.broken\_barh([(start, 1)], (10 \* (pid) - 5, 9), facecolors=(colors[(pid - 1) % len(colors)]))

    handles = [mpatches.Patch(color=colors[i % len(colors)], label=f'P{i + 1}') for i in range(len(unique\_processes))]

    plt.legend(handles=handles)

    plt.title("Gantt Chart")

    plt.show()

def preemptive\_priority(processes):

    n = len(processes)

    # Sort processes by arrival time and then by priority

    processes.sort(key=lambda x: (x[1], x[3]))

    current\_time = 0

    finish\_times = [0] \* n

    remaining\_burst = [burst for \_, \_, burst, \_ in processes]

    timeline = []  # For Gantt chart

    while True:

        # Find the process with the highest priority that has arrived

        idx = -1

        highest\_priority = float('inf')

        for i, (pid, arrival, burst, priority) in enumerate(processes):

            if arrival <= current\_time and remaining\_burst[i] > 0 and priority < highest\_priority:

                highest\_priority = priority

                idx = i

        if idx == -1:

            # If no process is ready, increment time

            current\_time += 1

            continue

        # Run the selected process for 1 time unit

        remaining\_burst[idx] -= 1

        timeline.append((processes[idx][0], current\_time))

        if remaining\_burst[idx] == 0:

            finish\_times[idx] = current\_time + 1

        current\_time += 1

        # If the process has finished, we might need to check for others

        if remaining\_burst[idx] == 0:

            for i in range(n):

                if remaining\_burst[i] == 0 and finish\_times[i] == 0:

                    finish\_times[i] = current\_time

        # Check if all processes are finished

        if all(burst == 0 for burst in remaining\_burst):

            break

    # Calculate turnaround and waiting times

    turnaround\_times = [finish\_times[i] - processes[i][1] for i in range(n)]

    waiting\_times = [turnaround\_times[i] - processes[i][2] for i in range(n)]

    # Print table

    print("\nProcess\tArrival\tBurst\tPriority\tFinish\tTurnaround\tWaiting")

    for i, (pid, arrival, burst, priority) in enumerate(processes):

        print(f"P{pid}\t{arrival}\t{burst}\t{priority}\t\t{finish\_times[i]}\t{turnaround\_times[i]}\t\t{waiting\_times[i]}")

    plot\_gantt\_chart(processes, timeline)

# Example Input

processes = [(1, 0, 4, 2), (2, 1, 3, 1), (3, 2, 1, 4), (4, 3, 2, 3)]

preemptive\_priority(processes)  
  
  
  
bankers algo:

class BankersAlgorithm:

    def \_\_init\_\_(self, processes, available, max\_need, allocation):

        self.processes = processes

        self.available = available

        self.max\_need = max\_need

        self.allocation = allocation

        self.need = [[max\_need[i][j] - allocation[i][j] for j in range(len(available))] for i in range(processes)]

    def is\_safe(self):

        work = self.available[:]

        finish = [False] \* self.processes

        safe\_sequence = []

        while len(safe\_sequence) < self.processes:

            found = False

            for i in range(self.processes):

                if not finish[i] and all(self.need[i][j] <= work[j] for j in range(len(work))):

                    work = [work[j] + self.allocation[i][j] for j in range(len(work))]

                    safe\_sequence.append(i)

                    finish[i] = True

                    found = True

                    break

            if not found:

                return False, []

        return True, safe\_sequence

# Example usage

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

    processes = 5

    available = [3, 3, 2]

    max\_need = [

        [7, 5, 3],

        [3, 2, 2],

        [9, 0, 2],

        [2, 2, 2],

        [4, 3, 3]

    ]

    allocation = [

        [0, 1, 0],

        [2, 0, 0],

        [3, 0, 2],

        [2, 1, 1],

        [0, 0, 2]

    ]

    banker = BankersAlgorithm(processes, available, max\_need, allocation)

    is\_safe, sequence = banker.is\_safe()

    if is\_safe:

        print("The system is in a safe state.")

        print(f"Safe Sequence: {sequence}")

    else:

        print("The system is not in a safe state.")

philosopher:  
#dining philosophers

import threading

import random

import time

NUM\_PHILOSOPHERS = 5

MAX\_EATS = 3

forks = [threading.Lock() for \_ in range(NUM\_PHILOSOPHERS)]

print\_lock = threading.Lock()

def eat(philosopher\_id):

left\_fork = philosopher\_id

right\_fork = (philosopher\_id + 1) % NUM\_PHILOSOPHERS

with forks[left\_fork], forks[right\_fork]:

with print\_lock:

print(f"Philosopher {philosopher\_id} is eating.")

time.sleep(random.uniform(1, 2))

with print\_lock:

print(f"Philosopher {philosopher\_id} finished eating.")

def philosopher(philosopher\_id):

eat\_count = 0

while eat\_count < MAX\_EATS:

with print\_lock:

print(f"Philosopher {philosopher\_id} is thinking.")

time.sleep(random.uniform(1, 2))

eat(philosopher\_id)

eat\_count += 1

with print\_lock:

print(f"Philosopher {philosopher\_id} has finished eating {MAX\_EATS} times.")

def main():

threads = []

for i in range(NUM\_PHILOSOPHERS):

thread = threading.Thread(target=philosopher, args=(i,))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

print("All philosophers have finished eating.")

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

main()