

| **TITLE:** Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority ) |
| --- |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** To implement basic Process management algorithms ( Round Robin,SRTN, Priority)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 2.** To understand the concept of process, thread and resource management.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems” McGraw Hill Third**

**Edition.**

1. **William Stallings, “Operating System Internal & Design Principles”, Pearson.**
2. **Andrew S. Tanenbaum, “Modern Operating System”, Prentice Hall.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests and a small number of processes with long CPU bursts. To provide good time-sharing performance, we may preempt a running process to let another one run. The ready list, also known as a run queue, in the operating system keeps a list of all processes that are ready to run and not blocked on some I/O or other system request, such as a semaphore. Then entries in this list are pointers to the process control block, which stores all information and state about a process.

When an I/O request for a process is complete, the process moves from the *waiting* state to the *ready* state and gets placed on the run queue.

The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

1. The current process goes from the *running* to the *waiting* state because it issues an I/O request or some operating system request that cannot be satisfied immediately.
2. The current process terminates.
3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the *running* to the *ready* state.
4. An I/O operation is complete for a process that requested it and the process now moves from the *waiting* to the*ready* state. The scheduler may then decide to preempt the currently-running process and move this *ready* process into the *running* state.

The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the scheduling algorithm (or scheduling policy). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

1. Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
2. Be efficient – keep the CPU busy all the time.
3. Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
4. Minimize response time – interactive users should see good performance
5. Minimize overhead – don’t waste too many resources. Keep scheduling time and context switch time at a minimum.
6. Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We’ll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more important, processes cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.
7. Avoid indefinite postponement – every process should get a chance to run eventually.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Description of the application to be implemented**:

**Round Robin Algorithm**

Round Robin (RR) Algorithm:

* Round Robin is a preemptive scheduling algorithm that assigns a fixed time slot (quantum) to each process in a circular order.
* Each process gets the CPU for a time quantum. If the process is not completed in one time quantum, it's moved to the end of the ready queue.
* This algorithm ensures fairness as each process gets an equal share of CPU time.
* It's simple to implement and provides good response time for short processes.

# Shortest Remaining Time First Algorithm :

SRTF is a preemptive version of the Shortest Job First (SJF) algorithm.

It selects the process with the shortest remaining burst time to execute next.

If a new process arrives with a shorter burst time than the remaining time of the current process, the current process is preempted.

This algorithm minimizes average waiting time but can lead to starvation of longer processes.

# Priority scheduling:

Priority Scheduling:

* In Priority Scheduling, each process is assigned a priority.
* The process with the highest priority (usually denoted by the lowest priority number) is executed first.
* It can be either preemptive or non-preemptive.
* In the preemptive version (which is implemented in our code), if a higher priority process arrives, the current process is preempted.
* This algorithm can lead to starvation of low-priority processes if not implemented with care.

**Implementation details:**

import sys

def round\_robin(n, at, bt, tq):

rem = bt.copy()

wt = [0] \* n

tat = [0] \* n

curr\_t = 0

queue = []

while any(rem):

for i in range(n):

if at[i] <= curr\_t and rem[i] > 0 and i not in queue:

queue.append(i)

if queue:

curr = queue.pop(0)

if rem[curr] > tq:

curr\_t += tq

rem[curr] -= tq

queue.append(curr)

else:

curr\_t += rem[curr]

tat[curr] = curr\_t - at[curr]

wt[curr] = tat[curr] - bt[curr]

rem[curr] = 0

else:

curr\_t += 1

print\_results(n, at, bt, wt, tat)

def srtf(n, at, bt):

rem = bt.copy()

wt = [0] \* n

tat = [0] \* n

curr\_t = 0

completed = 0

while completed < n:

shortest = -1

shortest\_time = sys.maxsize

for i in range(n):

if at[i] <= curr\_t and rem[i] > 0:

if rem[i] < shortest\_time:

shortest = i

shortest\_time = rem[i]

if shortest == -1:

curr\_t += 1

else:

rem[shortest] -= 1

curr\_t += 1

if rem[shortest] == 0:

completed += 1

tat[shortest] = curr\_t - at[shortest]

wt[shortest] = tat[shortest] - bt[shortest]

print\_results(n, at, bt, wt, tat)

def priority\_based(n, at, bt, priority):

rem = bt.copy()

wt = [0] \* n

tat = [0] \* n

curr\_t = 0

completed = 0

while completed < n:

highest\_priority = -1

highest\_priority\_value = sys.maxsize

for i in range(n):

if at[i] <= curr\_t and rem[i] > 0:

if priority[i] < highest\_priority\_value:

highest\_priority = i

highest\_priority\_value = priority[i]

if highest\_priority == -1:

curr\_t += 1

else:

rem[highest\_priority] -= 1

curr\_t += 1

if rem[highest\_priority] == 0:

completed += 1

tat[highest\_priority] = curr\_t - at[highest\_priority]

wt[highest\_priority] = tat[highest\_priority] - bt[highest\_priority]

print\_results(n, at, bt, wt, tat, priority)

def print\_results(n, at, bt, wt, tat, priority=None):

awt = sum(wt) / n

atat = sum(tat) / n

if priority:

print("Process\t\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time")

for i in range(n):

print(f"{i+1}\t\t{at[i]}\t\t{bt[i]}\t\t{priority[i]}\t\t{wt[i]}\t\t{tat[i]}")

else:

print("Process\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time")

for i in range(n):

print(f"{i+1}\t\t{at[i]}\t\t{bt[i]}\t\t{wt[i]}\t\t{tat[i]}")

print(f"\nAverage Waiting Time: {awt:.2f}")

print(f"Average Turnaround Time: {atat:.2f}")

def get\_input():

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

at = []

bt = []

priority = []

for i in range(n):

arrival\_time = int(input(f"Enter arrival time for process {i+1}: "))

burst\_time = int(input(f"Enter burst time for process {i+1}: "))

priority\_val = int(input(f"Enter priority for process {i+1} (lower value means higher priority): "))

at.append(arrival\_time)

bt.append(burst\_time)

priority.append(priority\_val)

return n, at, bt, priority

def main():

n, at, bt, priority = get\_input()

while True:

print("\nAvailable scheduling algorithms:")

print("1. Round Robin")

print("2. Shortest Remaining Time First (SRTF)")

print("3. Priority-based")

print("4. Exit")

choice = input("Enter your choice (1-4): ")

if choice == '1':

tq = int(input("Enter the time quantum: "))

round\_robin(n, at, bt, tq)

elif choice == '2':

srtf(n, at, bt)

elif choice == '3':

priority\_based(n, at, bt, priority)

elif choice == '4':

print("Exiting the program.")

break

else:

print("Invalid choice. Please try again.")

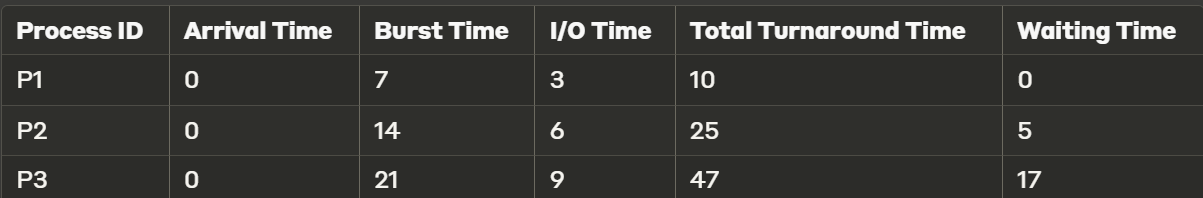
if \_\_name\_\_ == "\_\_main\_\_":

main()

**Conclusion:** Learnt and implemented pre-emptive process scheduling algorithms.

**Post Lab Descriptive Questions**

1. Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?



Total execution time: 47 ms

CPU idle time: 2 ms at the beginning (0-2) and 3 ms at the end (44-47), totaling 5 ms

Percentage of CPU idle time: (5 / 47) \* 100 ≈ 10.64%

1. What is Starvation?

Starvation in operating systems refers to a situation where a process is indefinitely denied access to necessary resources, preventing it from completing its task. This phenomenon typically occurs in priority-based scheduling systems, where lower-priority processes are continually bypassed in favor of higher-priority ones. Unlike deadlock, where processes are caught in a circular wait, starvation can persist even when resources are available but not fairly allocated. It often results from inefficient scheduling algorithms that consistently favor certain processes, leading to poor system performance and unfairness. Common examples include low-priority processes never running in priority scheduling or longer jobs being perpetually preempted in Shortest Job First scheduling. To mitigate starvation, systems may employ techniques such as aging (gradually increasing the priority of waiting processes), fair scheduling algorithms like Round Robin, or setting maximum resource allocation times. Understanding and preventing starvation is crucial in developing equitable and efficient operating systems.

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**