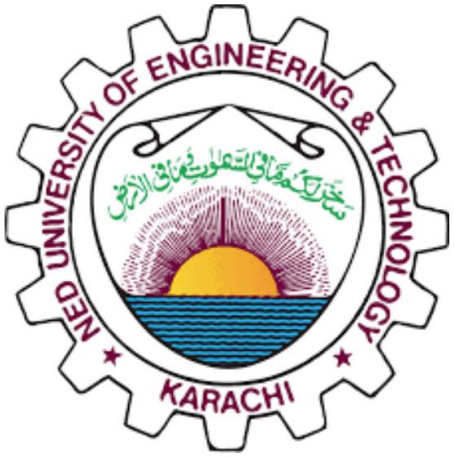


Complex Engineering Problem

Submitted By

Muhammad Ayan Tahir SE-22035

Muhammad Bilal SE-22037

Cloud Data Backup Scheduling

Course: Operating Systems (SE-303)

Submitted to: Miss. Shamama

Cloud Data Backup Scheduling

ABSTRACT

The effectiveness of the cloud data backup scheduling is important as it helps to ease the downtime, avoid resource wastage, and ensure that the important files are well secured in the required time. The focus of this study is on the ever-increasing demand of reliable backup systems for cloud hosted content. In line with this, it seeks to establish operating parameters for cloud organizations that will help to achieve optimized goals for their backup systems. In this report, we rank the First Come First Served, Shortest Job First – both preemptive and non-preemptive, Priority Scheduling –both preemptive and non-preemptive and round robin techniques according to the requirements and goals of cloud backup operations Furthermore, it seeks to enhance existing methods and propose new effective and efficient scheduling techniques that would be universal in order to achieve a goal of occupying a single higher layer of cloud scheduling. This goal is attributable to factors such as fault tolerance, scalability and adaptability to diverse workload patterns. This study offers practical suggestions for advancement of practices of data backup in cloud environments that are volatile and high volume.

INTRODUCTION

Cloud data backup scheduling is the process of determining the time and the order in which the backup processes will be carried out in the storage which is provided by the cloud. In the recent past, there has been a dramatic increase in organizations using cloud databases and storage as their primary business storage. Many companies' databases are exclusively hosted on cloud databases such as AWS (amazon web services), Microsoft's Azure and google cloud. Due to this reason, effective scheduling of each data package is crucial and more relevant than ever.

In high-traffic cloud systems, such as those supporting real-time analytics platforms or global enterprise applications, the complexity of scheduling backup tasks increases significantly. These systems must handle backups for large volumes of data while balancing competing priorities such as performance, storage availability, and cost-efficiency. Effective scheduling ensures timely data snapshots, reduces recovery point objectives (RPOs), and maintains overall system reliability.

Task Distribution

Ayan Tahir(SE-22035)	Muhammad Bilal(SE-22037)
Round robin, research work, priority preemptive.	FCFS, SJF, priority non-preemptive.

SCENARIO

Consider a scenario, where 5 different data backup requests were made by different users/organizations. Due to high traffic, each request was within minutes of each other. Some requests can have higher priority such as those coming from healthcare organizations or simply from premium users. The data backup rate is **1 GB/minute** which is assumed to be the unit of burst time for each request i.e. 1 GB of data requires 1 minute to backup, so the burst time is 1 minute and consequently, 6 GBs of data requires 6 minutes so burst time is 6 minutes and so on. The requests' burst times (in minutes) and arrival times (in minutes) are given as follows. The requests are named as "Jobs" or "processes" interchangeably for simplicity.

The Jobs Simulated: (larger number is a higher priority)

Job Id	Arrival Time(m)	Burst time(m)	Priority
Job1	2	4	3
Job2	7	6	1
Job3	0	7	4
Job4	4	10	2
Job5	10	20	0

Now, we simulate each Job using First Come First Served, Shortest Job First – both preemptive and non-preemptive, Priority Scheduling –both preemptive and non-preemptive and round robin algorithms in **C** language and then compare them to analyze which scheduling is best and most feasible. We also give detailed information about each algorithm such as turnaround time, waiting time, fairness, starvation, implementation overhead, and context switching overhead. If you are unfamiliar with these terms, following are brief explanations,

Turnaround Time: Total time taken to complete a task from arrival to finish. Lower TAT indicates faster task completion.

Waiting Time: Time spent waiting in the ready queue before execution. Lower WT improves system responsiveness.

Fairness: Equal opportunity for tasks to access CPU/Network resources.

Starvation: Long delays or indefinite postponement for certain tasks due to scheduling preferences.

Implementation Overhead: The complexity of implementing and managing the scheduling.

Context Switching Overhead: The time and resources used to switch between tasks during preemptive scheduling.

1) FIRST COME FIRST SERVED SCHEDULING ALGORITHM

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue**[1]**. In our scenario's context , If job 1 arrives at 0 minutes and job 2 arrives at 3 minutes. Job 2 won't start being backed till job 1 is completely finished (non-preemption). Also, consider that jobs come in this order: job 1, job 2, job 3. They will also be backed up in this exact same order. Hence first come first served. Now lets implement our scenario with this algorithm.

SOURCE CODE:

```
#include<stdio.h>
struct Process
{
  int pid;
  int waitingTime;
  int arrivalTime;
  int backupSize;
  int turnaroundTime;
  int startingTime;
} p[10];
void calculateTimes(struct Process p[], int n)
{
  int currentTime = 0;
  for (int i = 0; i < n; i++)
  {
    if (currentTime < p[i].arrivalTime)
    {
      currentTime = p[i].arrivalTime;
    }
    p[i].startingTime = currentTime;
    p[i].turnaroundTime = (p[i].startingTime + p[i].backupSize) -
p[i].arrivalTime;
    p[i].waitingTime = p[i].turnaroundTime - p[i].backupSize;
    currentTime += p[i].backupSize;
  }
}
void printTable(struct Process p[], int n) {
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting
Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid,
p[i].arrivalTime, p[i].backupSize, p[i].waitingTime,
p[i].turnaroundTime);
        totalWaitingTime += p[i].waitingTime;
        totalTurnaroundTime += p[i].turnaroundTime;
    }

    float avgWaitingTime = (float)totalWaitingTime / n;
    float avgTurnaroundTime = (float)totalTurnaroundTime / n;

    printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
    printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}
int main()
{
  int n;
  printf("Enter the number of process\n");
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
  {
    printf("Enter the arrival time for process %d: ", i + 1);
    scanf("%d", &p[i].arrivalTime);
    printf("Enter the back up size for process %d (in GB): ", i + 1);
    scanf("%d", &p[i].backupSize);
    p[i].pid = i + 1;
  }
  // sorting all processes
  for (int i = 0; i < n - 1; i++)
  {
    for (int j = 0; j < n - i - 1; j++)
    {
      if (p[j].arrivalTime > p[j + 1].arrivalTime)
      {
        struct Process temp = p[j];
        p[j] = p[j + 1];
        p[j + 1] = temp;
      }
    }
  }

  calculateTimes(p, n);
  printTable(p, n);

  return 0;
}
```

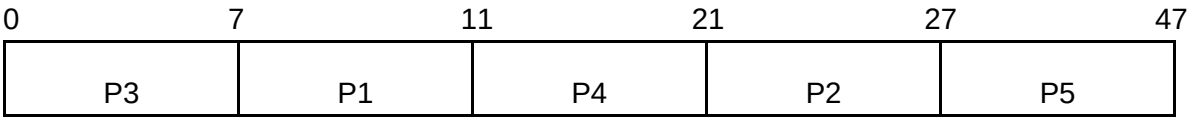
OUTPUT:

```
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$ ./fcfs
Enter the number of process
5
Enter the arrival time for process 1: 2
Enter the back up size for process 1 (in GB): 4
Enter the arrival time for process 2: 7
Enter the back up size for process 2 (in GB): 6
Enter the arrival time for process 3: 0
Enter the back up size for process 3 (in GB): 7
Enter the arrival time for process 4: 4
Enter the back up size for process 4 (in GB): 10
Enter the arrival time for process 5: 10
Enter the back up size for process 5 (in GB): 20

Process Arrival Time    Backup size     Waiting Time    Turnaround Time
P3              0               7               0               7
P1              2               4               5               9
P4              4               10              7               17
P2              7               6               14              20
P5              10              20              17              37

Average Waiting Time: 8.60 m
Average Turnaround Time: 18.00 m
```

GANTT CHART:



2) SHORTEST JOB FIRST SCHEDULING ALGORITHM (SJF)

The SJF algorithm is a special case of priority scheduling. Each process is equipped with a priority number that is burst time. The CPU is allocated to the process that has the highest priority. If several processes have the same priority, then it will use the FCFS algorithm. Scheduling priority consists of two schemes, non-preemptive and preemptive**[2]**. In our context, in this scheduling approach, the jobs that require less burst time i.e. the data packages that are smaller in size (less GBs) are preferred or prioritized over larger ones. It may be preemptive (current backup job is interrupted to switch to a newly arrived smaller backup job) or non-preemptive (once backup job is being processed, it can't be interrupted).

- **NON-PREEMPTIVE APPROACH**

SOURCE CODE:

```
#include <stdio.h>

struct BackupJob
{
  int jobId;
  int waitingTime;
  int arrivalTime;
  int backupSize; // Equivalent to burst time
  int turnaroundTime;
  int completionTime;
  int remainingSize; // Remaining data size to backup
} jobs[10];

void printJobTable(struct BackupJob jobs[], int n)
{
  int totalWaitingTime = 0;
  int totalTurnaroundTime = 0;

  printf("\nJob\tStart Time\tBackup Size\tWaiting Time\tTurnaround
Time\tCompletion Time\n");
  for (int i = 0; i < n; i++)
  {
    printf("J%d\t\t%d\t\t%d GB\t\t%d\t\t%d\t\t%d\n", jobs[i].jobId,
jobs[i].arrivalTime, jobs[i].backupSize, jobs[i].waitingTime,
jobs[i].turnaroundTime, jobs[i].completionTime);
    totalWaitingTime += jobs[i].waitingTime;
    totalTurnaroundTime += jobs[i].turnaroundTime;
  }

  float avgWaitingTime = (float)totalWaitingTime / n;
  float avgTurnaroundTime = (float)totalTurnaroundTime / n;

  printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
  printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}

void calculateBackupTimes(struct BackupJob jobs[], int n)
{
  int completedJobs = 0;
  int currentTime = 0;
  while (completedJobs < n)
```

```
  {
    int shortestJob = -1;
    for (int i = 0; i < n; i++)
    {
      if (jobs[i].arrivalTime <= currentTime && jobs[i].remainingSize
> 0)
      {
        if (shortestJob == -1 || jobs[i].backupSize <
jobs[shortestJob].backupSize)
        {
          shortestJob = i;
        }
      }
    }
    if (shortestJob != -1)
    {
      currentTime += jobs[shortestJob].backupSize;
      jobs[shortestJob].completionTime = currentTime;
      jobs[shortestJob].turnaroundTime =
jobs[shortestJob].completionTime - jobs[shortestJob].arrivalTime;
      jobs[shortestJob].waitingTime = jobs[shortestJob].turnaroundTime
- jobs[shortestJob].backupSize;
      jobs[shortestJob].remainingSize = 0;

      completedJobs++;
    }
    else
    {
      currentTime++;
    }
  }
}

int main()
{
  int n;
  printf("Enter the number of backup jobs: ");
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
  {
    printf("Enter the arrival time for backup job %d: ", i + 1);
    scanf("%d", &jobs[i].arrivalTime);
    printf("Enter the backup size for job %d (in GB): ", i + 1);
    scanf("%d", &jobs[i].backupSize);
    jobs[i].jobId = i + 1;
```

```
        jobs[i].remainingSize = jobs[i].backupSize;
    }

    calculateBackupTimes(jobs, n);
    printJobTable(jobs, n);

    return 0;
}
```

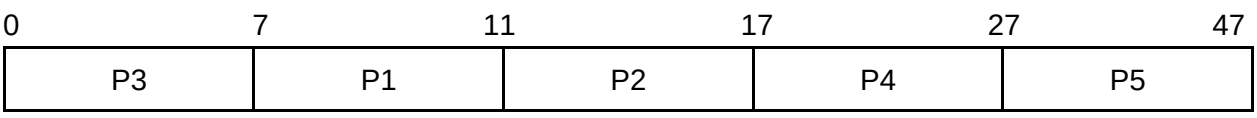
OUTPUT:

```
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$ ./sjf_np
Enter the number of backup jobs: 5
Enter the arrival time for backup job 1: 2
Enter the backup size for job 1 (in GB): 4
Enter the arrival time for backup job 2: 7
Enter the backup size for job 2 (in GB): 6
Enter the arrival time for backup job 3: 0
Enter the backup size for job 3 (in GB): 7
Enter the arrival time for backup job 4: 4
Enter the backup size for job 4 (in GB): 10
Enter the arrival time for backup job 5: 10
Enter the backup size for job 5 (in GB): 20

Job     Start Time      Backup Size     Waiting Time    Turnaround Time Completion Time
J1              2               4 GB            5               9               11
J2              7               6 GB            4               10              17
J3              0               7 GB            0               7               7
J4              4               10 GB           13              23              27
J5              10              20 GB           17              37              47

Average Waiting Time: 7.80 m
Average Turnaround Time: 17.20 m
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$
```

GANTT CHART:



- **PREEMPTIVE APPROACH**

SOURCE CODE:

```
#include <stdio.h>

struct BackupJob
{
  int jobId;
  int waitingTime;
  int arrivalTime;
  int backupSize;
  int turnaroundTime;
  int completionTime;
  int remainingSize;
  int executionTime;
} jobs[10];

void printJobTable(struct BackupJob jobs[], int n)
{
  int totalWaitingTime = 0;
  int totalTurnaroundTime = 0;

  printf("\nJob\tStart Time\tBackup Size\tWaiting Time\tTurnaround
Time\tCompletion Time\n");
  for (int i = 0; i < n; i++)
  {
    printf("J%d\t\t%d\t\t%d GB\t\t%d\t\t%d\t\t%d\n", jobs[i].jobId,
jobs[i].arrivalTime, jobs[i].backupSize, jobs[i].waitingTime,
jobs[i].turnaroundTime, jobs[i].completionTime);
    totalWaitingTime += jobs[i].waitingTime;
    totalTurnaroundTime += jobs[i].turnaroundTime;
  }

  float avgWaitingTime = (float)totalWaitingTime / n;
  float avgTurnaroundTime = (float)totalTurnaroundTime / n;

  printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
  printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}

void calculateBackupTimes(struct BackupJob jobs[], int n)
{
```

```
  int completedJobs = 0;
  for (int time = 0; completedJobs != n; time++)
  {
    int smallest = -1;
    for (int i = 0; i < n; i++)
    {
      if (jobs[i].arrivalTime <= time && jobs[i].remainingSize > 0)
      {
        if (smallest == -1 || jobs[i].remainingSize <
jobs[smallest].remainingSize)
        {
          smallest = i;
        }
      }
    }

    if (smallest != -1)
    {
      jobs[smallest].remainingSize--;
      jobs[smallest].executionTime++;

      if (jobs[smallest].remainingSize == 0)
      {
        completedJobs++;
        jobs[smallest].completionTime = time + 1;
      }
    }
  }

  for (int i = 0; i < n; i++)
  {
    jobs[i].turnaroundTime = jobs[i].completionTime -
jobs[i].arrivalTime;
    jobs[i].waitingTime = jobs[i].completionTime -
jobs[i].executionTime - jobs[i].arrivalTime;
  }
}
int main()
{
  int n;
  printf("Enter the number of backup jobs: ");
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
  {
    printf("Enter the arrival time for backup job %d: ", i + 1);
```

```
    scanf("%d", &jobs[i].arrivalTime);
    printf("Enter the backup size for job %d (in GB): ", i + 1);
    scanf("%d", &jobs[i].backupSize);
    jobs[i].jobId = i + 1;
    jobs[i].remainingSize = jobs[i].backupSize;
  }

  calculateBackupTimes(jobs, n);
  printJobTable(jobs, n);

  return 0;
}
```

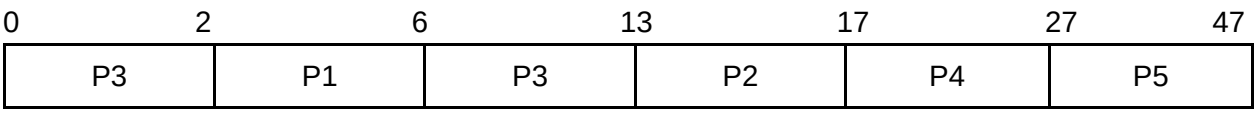
OUTPUT:

```
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$ ./sjf
Enter the number of backup jobs: 5
Enter the arrival time for backup job 1: 2
Enter the backup size for job 1 (in GB): 4
Enter the arrival time for backup job 2: 7
Enter the backup size for job 2 (in GB): 6
Enter the arrival time for backup job 3: 0
Enter the backup size for job 3 (in GB): 7
Enter the arrival time for backup job 4: 4
Enter the backup size for job 4 (in GB): 10
Enter the arrival time for backup job 5: 10
Enter the backup size for job 5 (in GB): 20

Job     Start Time      Backup Size     Waiting Time    Turnaround Time Completion Time
J1              2               4 GB            0               4               6
J2              7               6 GB            4               10              17
J3              0               7 GB            4               11              11
J4              4               10 GB           13              23              27
J5              10              20 GB           17              37              47

Average Waiting Time: 7.60 m
Average Turnaround Time: 17.00 m
```

GANTT CHART:



3) PRIORITY BASED SCHEDULING ALGORITHM

In this scheduling approach, A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order**[1]**. For example, in our scenario, backup requests or “jobs” from a critical healthcare organization or a job from a premium cloud user might be prioritized or “preferred” over others less critical ones. This scheduling might be preemptive (current job will be interrupted for a newly arrived higher priority job) or non preemptive (current job won’t be interrupted).

- **NON-PREEMPTIVE APPROACH**

SOURCE CODE:

```
#include <stdio.h>
struct BackupJob
{
  int pid;
  int waitingTime;
  int arrivalTime;
  int backupSize;
  int turnaroundTime;
  int remainingTime;
  int completionTime;
  int priority;
} jobs[10];

void printTable(struct BackupJob jobs[], int n)
{
  int totalWaitingTime = 0;
  int totalTurnaroundTime = 0;

  printf("\nProcess\tArrival Time\tBack upspace\tWaiting
Time\tTurnaround Time\tCompletion Time\n");
  for (int i = 0; i < n; i++)
  {
    printf("jobs%d\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n", jobs[i].pid,
jobs[i].arrivalTime, jobs[i].backupSize, jobs[i].waitingTime,
jobs[i].turnaroundTime, jobs[i].completionTime);
    totalWaitingTime += jobs[i].waitingTime;
    totalTurnaroundTime += jobs[i].turnaroundTime;
  }

  float avgWaitingTime = (float)totalWaitingTime / n;
  float avgTurnaroundTime = (float)totalTurnaroundTime / n;
```

```
  printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
  printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}

void calculateTimes(struct BackupJob jobs[], int n)
{
  int completedJobs = 0;
  int currentTime = 0;
  while (completedJobs < n)
  {
    int highestPriority = -1;
    for (int i = 0; i < n; i++)
    {
      if (jobs[i].arrivalTime <= currentTime && jobs[i].remainingTime
> 0)
      {
        if (highestPriority == -1 || jobs[i].priority <=
jobs[highestPriority].priority)
        {
          highestPriority = i;
        }
      }
    }
    if (highestPriority != -1)
    {
      currentTime += jobs[highestPriority].backupSize;
      jobs[highestPriority].completionTime = currentTime;
      jobs[highestPriority].turnaroundTime =
jobs[highestPriority].completionTime -
jobs[highestPriority].arrivalTime;
      jobs[highestPriority].waitingTime =
jobs[highestPriority].turnaroundTime -
jobs[highestPriority].backupSize;
      jobs[highestPriority].remainingTime = 0;
      completedJobs++;
    }
    else
    {
      currentTime++;
    }
  }
}
int main()
{
  int n;
```

```
  printf("Enter the number of process\n");
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
  {
    printf("Enter the arrival time for process %d: ", i + 1);
    scanf("%d", &jobs[i].arrivalTime);
    printf("Enter the back up space for process %d (in GB): (in GB)",
i + 1);
    scanf("%d", &jobs[i].backupSize);
    printf("Enter the priority for process %d: ", i + 1);
    scanf("%d", &jobs[i].priority);
    jobs[i].pid = i + 1;
    jobs[i].remainingTime = jobs[i].backupSize;
  }
  calculateTimes(jobs, n);

  printTable(jobs, n);

  return 0;
}
```

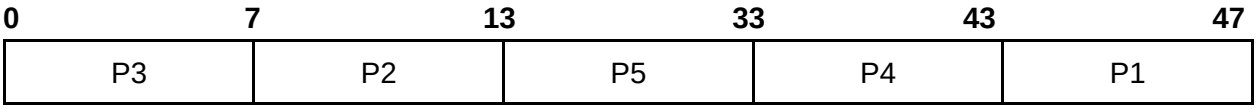
OUTPUT:

```
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$ ./prority_np
Enter the number of process
5
Enter the arrival time for process 1: 2
Enter the back up space for process 1 (in GB): (in GB)4
Enter the priority for process 1: 3
Enter the arrival time for process 2: 7
Enter the back up space for process 2 (in GB): (in GB)6
Enter the priority for process 2: 1
Enter the arrival time for process 3: 0
Enter the back up space for process 3 (in GB): (in GB)7
Enter the priority for process 3: 4
Enter the arrival time for process 4: 4
Enter the back up space for process 4 (in GB): (in GB)10
Enter the priority for process 4: 2
Enter the arrival time for process 5: 10
Enter the back up space for process 5 (in GB): (in GB)20
Enter the priority for process 5: 0

Process Arrival Time    Back upspace    Waiting Time    Turnaround Time Completion Time
jobs1           2               4               41              45      47
jobs2           7               6               0               6       13
jobs3           0               7               0               7       7
jobs4           4               10              29              39      43
jobs5           10              20              3               23      33

Average Waiting Time: 14.60 m
Average Turnaround Time: 24.00 m
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$
```

GANTT CHART:



- **PREEMPTIVE APPROACH**

SOURCE CODE:

```
#include <stdio.h>
struct BackupJob
{
  int pid;
  int waitingTime;
  int arrivalTime;
  int backupSize;
  int turnaroundTime;
  int remainingTime;
  int completionTime;
  int priority;
} jobs[10];

void printTable(struct BackupJob jobs[], int n)
{
  int totalWaitingTime = 0;
  int totalTurnaroundTime = 0;

  printf("\nProcess\tArrival Time\tBack upspace\tWaiting
Time\tTurnaround Time\tCompletion Time\n");
  for (int i = 0; i < n; i++)
  {
    printf("jobs%d\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n", jobs[i].pid,
jobs[i].arrivalTime, jobs[i].backupSize, jobs[i].waitingTime,
jobs[i].turnaroundTime, jobs[i].completionTime);
    totalWaitingTime += jobs[i].waitingTime;
    totalTurnaroundTime += jobs[i].turnaroundTime;
  }

  float avgWaitingTime = (float)totalWaitingTime / n;
  float avgTurnaroundTime = (float)totalTurnaroundTime / n;

  printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
```

```
  printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}

void calculateTimes(struct BackupJob jobs[], int n)
{
  int remain = 0;
  for (int time = 0; remain != n; time++)
  {
    int highestPriority = -1;
    for (int i = 0; i < n; i++)
    {
      if (jobs[i].arrivalTime <= time && jobs[i].remainingTime > 0)
      {
        if (highestPriority == -1 || jobs[i].priority <=
jobs[highestPriority].priority)
        {
          highestPriority = i;
        }
      }
    }
    jobs[highestPriority].remainingTime--;

    if (jobs[highestPriority].remainingTime == 0)
    {
      remain++;
      jobs[highestPriority].completionTime = time + 1;
    }
  }
  for (int i = 0; i < n; i++)
  {
    jobs[i].turnaroundTime = jobs[i].completionTime -
jobs[i].arrivalTime;
    jobs[i].waitingTime = jobs[i].turnaroundTime - jobs[i].backupSize;
  }
}
int main()
{
  int n;
  printf("Enter the number of process\n");
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
  {
    printf("Enter the arrival time for process %d: ", i + 1);
    scanf("%d", &jobs[i].arrivalTime);
```

```
    printf("Enter the back up space for process %d (in GB): (in GB)",
i + 1);
    scanf("%d", &jobs[i].backupSize);
    printf("Enter the priority for process %d: ", i + 1);
    scanf("%d", &jobs[i].priority);
    jobs[i].pid = i + 1;
    jobs[i].remainingTime = jobs[i].backupSize;
  }
  calculateTimes(jobs, n);

  printTable(jobs, n);

  return 0;
}
```

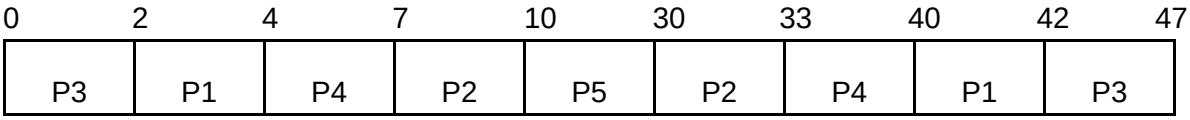
OUTPUT:

```
Enter the number of process
5
Enter the arrival time for process 1: 2
Enter the back up space for process 1 (in GB):4
Enter the priority for process 1: 3
Enter the arrival time for process 2: 7
Enter the back up space for process 2 (in GB):6
Enter the priority for process 2: 1
Enter the arrival time for process 3: 0
Enter the back up space for process 3 (in GB):7
Enter the priority for process 3: 4
Enter the arrival time for process 4: 4
Enter the back up space for process 4 (in GB):10
Enter the priority for process 4: 2
Enter the arrival time for process 5: 10
Enter the back up space for process 5 (in GB):20
Enter the priority for process 5: 0

Process Arrival Time    Back upspace    Waiting Time    Turnaround Time Completion Time
jobs1           2               4               36              40      42
jobs2           7               6               20              26      33
jobs3           0               7               40              47      47
jobs4           4               10              26              36      40
jobs5           10              20              0               20      30

Average Waiting Time: 24.40 m
Average Turnaround Time: 33.80 m
muhammad-bilal@muhammad-bilal-VirtualBox:~/os_cep$ nano priority_scheduling.c
```

GANTT CHART:



4) ROUND ROBIN SCHEDULING ALGORITHM (RR)

In the Round Robin concept, the algorithm uses time-sharing. The algorithm is the same as FCFS, but it is preempted. Each process gets CPU time called a quantum time to limit the processing time, typically 1-100 milliseconds. After time runs out, the process is delayed and added to the ready queue**[2]**. For example, in our scenario, if the time quantum is 1 minute, 'n' data backup jobs are given 1 minute each for round one and then again 1 minute each for round two and so on until they are completed. If new jobs arrive in between then they are placed in the queue to be executed in the next round(s).

SOURCE CODE:

```
#include <stdio.h>
struct BackupJob {
    int pid;
    int waitingTime;
    int arrivalTime;
    int backupSize;
    int turnaroundTime;
    int remainingTime;
    int completionTime;
} jobs[10];

void printTable(struct BackupJob jobs[], int n) {
    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    printf("\nProcess\tArrival Time\tBackup Size\tWaiting
Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("Job%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
               jobs[i].pid, jobs[i].arrivalTime, jobs[i].backupSize,
               jobs[i].waitingTime, jobs[i].turnaroundTime,
jobs[i].completionTime);
        totalWaitingTime += jobs[i].waitingTime;
        totalTurnaroundTime += jobs[i].turnaroundTime;
```

```
    }

    float avgWaitingTime = (float)totalWaitingTime / n;
    float avgTurnaroundTime = (float)totalTurnaroundTime / n;

    printf("\nAverage Waiting Time: %.2f ms", avgWaitingTime);
    printf("\nAverage Turnaround Time: %.2f ms\n", avgTurnaroundTime);
}

void roundRobin(struct BackupJob jobs[], int n, int quantum) {
    int completedJobs = 0, currentTime = 0, executed;

    while (completedJobs < n) {
        executed = 0;

        for (int i = 0; i < n; i++) {
            if (jobs[i].arrivalTime <= currentTime &&
jobs[i].remainingTime > 0) {
                if (jobs[i].remainingTime <= quantum) {
                    currentTime += jobs[i].remainingTime;
                    jobs[i].completionTime = currentTime;
                    jobs[i].turnaroundTime = jobs[i].completionTime -
jobs[i].arrivalTime;
                    jobs[i].waitingTime = jobs[i].turnaroundTime -
jobs[i].backupSize;
                    jobs[i].remainingTime = 0;
                    completedJobs++;
                } else {
                    currentTime += quantum;
                    jobs[i].remainingTime -= quantum;
                }
                executed = 1;
            }
        }

        if (!executed) {
            currentTime++;
        }
    }

    printTable(jobs, n);
}

int main() {
    int n, quantum;
```

```c
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter the arrival time for process %d: ", i + 1);
        scanf("%d", &jobs[i].arrivalTime);
        printf("Enter the backup size for process %d (in GB): ", i +
1);
        scanf("%d", &jobs[i].backupSize);
        jobs[i].pid = i + 1;
        jobs[i].remainingTime = jobs[i].backupSize;
    }

    printf("Enter the time quantum: ");
    scanf("%d", &quantum);

    roundRobin(jobs, n, quantum);

    return 0;
}
```

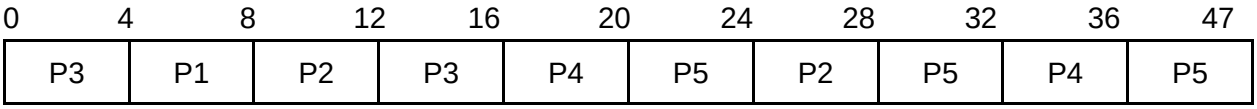
OUTPUT:

```
ayan@ayan:~$ ./roundRobin
Enter the number of processes: 5
Enter the arrival time for process 1: 2
Enter the backup size for process 1 (in GB): 4
Enter the arrival time for process 2: 7
Enter the backup size for process 2 (in GB): 6
Enter the arrival time for process 3: 0
Enter the backup size for process 3 (in GB): 7
Enter the arrival time for process 4: 4
Enter the backup size for process 4 (in GB): 10
Enter the arrival time for process 5: 10
Enter the backup size for process 5 (in GB): 20
Enter the time quantum: 4

Process Arrival Time    Backup Size     Waiting Time    Turnaround Time Completion Time
Job1            2               4               6               10              12
Job2            7               6               16              22              29
Job3            0               7               12              19              19
Job4            4               10              17              27              31
Job5            10              20              17              37              47

Average Waiting Time: 13.60 m
Average Turnaround Time: 23.00 m
ayan@ayan:~$
```

GANTT CHART:



COMPARISON OF THE ABOVE ALGORITHMS

Aspect	**FCFS**	**SJF Preemptive**	**SJF Non-Preemptive**	**Priority Preemptive**	**Priority Non-Preemptive**	**Round Robin**
Turnaround Time	High (due to waiting for long jobs)	Low (short jobs finish faster)	Low (minimizes waiting for short jobs)	Variable (based on priorities, could be high for low priority tasks)	Variable (can be high for low priority tasks)	Moderate (depends on time quantum)
Waiting Time	High (waiting for previous jobs to finish)	Low (jobs with shorter burst times wait less)	Low (jobs with shorter burst times wait less)	Variable (lower priority jobs wait longer)	Variable (lower priority jobs wait longer)	Low (due to time-sharing nature)
Fairness	Not very fair (long jobs dominate)	Less fair (short jobs always preempt long jobs)	Fairer (short jobs get processed without being preempted)	Less fair (based on priority)	Less fair (based on priority)	Fair (time slice distributes CPU equally)
Starvation	No starvation	Yes (long jobs may be starved if short jobs keep arriving)	No starvation (jobs are processed in order)	Yes (low priority jobs may be starved)	Yes (low priority jobs may be starved)	No starvation
Implementation Overhead	Low	Moderate (preemption requires context switching)	Low (no preemption)	High (preemption and priority calculation)	Moderate (priority calculation)	Moderate (context switching for time slices)

Context Switching Overhead	Low	High (due to frequent preemption)	Low (no preemption)	High (due to frequent preemption)	Low (no preemption)	Moderate (depends on time slice size)

DETAILED COMPARISON:

- FCFS (First-Come, First-Served): Simple to implement but can suffer from the convoy effect, where a long process delays all subsequent processes.
- SJF Preemptive (Shortest Job First, also known as Shortest Remaining Time First): Optimal in terms of minimizing average waiting time, but difficult to implement due to the need for precise knowledge or estimation of burst times. Can lead to starvation of long jobs.
- SJF Non-Preemptive: Similar to preemptive SJF but without interruption. It also minimizes average waiting time but can lead to starvation.
- Priority Preemptive: Processes are preempted based on priority. It allows important tasks to execute first but can cause starvation for lower-priority processes.
- Priority Non-Preemptive: Once a process starts, it runs to completion, even if a higher-priority process arrives during its execution. May result in starvation for lower-priority processes but avoids the overhead of preemption.
- Round Robin (RR): Designed for time-sharing systems. Each process receives a time slice (quantum) and cycles through the queue. This ensures fairness but can lead to high context-switching overhead if the time quantum is too small.

RECOMMENDATIONS:

In our scenario, the preferred scheduling algorithm should be non-preemptive. This is because if it were to be preemptive, data backup jobs would get frequently interrupted and as a result, the system might need to reanalyze and rescan how many chunks of data for each job have been uploaded before resuming that job.This is done so that the system knows from which point the job was interrupted and from which point the job should resume. This preemptive approach would result in very high Disk usage for the cloud server and consequently high CPU usage which might be very inefficient. Additionally, there might be priorities associated with each job such as those from critical organizations or users. So the best choice in this scenario might be the **Non-preemptive priority scheduling algorithm** as it is predictable, has low context switching, can accommodate priority and is overall best suited for our cloud backup system needs.

References:

[1] *Operating System Concepts* by Abraham Silberschatz, Peter B. Galvin, and Greg Gagne.
[2] *Comparison Analysis of CPU Scheduling : FCFS, SJF and Round Robin* by Andysah Putera Utama Siahaan.