1. **Program Statement**

This program simulates processor using a binary heap priority queue. It solves the problem of figuring out how many processors are needed to run four different types of jobs.

1. **Requirements**
   1. **Assumptions**
      1. Arrival and processing times are integers
      2. Working with 1 CPU for the first try
      3. Working with 2 CPU for the second try
   2. **Specifications**
      1. Regular jobs:
         1. A: Arrival time: 5 +/- 1. Processing time: 3 +/- 2
         2. B: Arrival time: 10 +/- 1. Processing time: 9 +/- 2
         3. C: Arrival time: 25 +/- 1. Processing time: 13 +/- 2
      2. Highest priority job:
         1. D: Arrival time: 30 +/- 5. Processing time: 10 +/- 2
      3. Random number to be generated for each job’s arrival time and processing time for each time.
      4. Track time
      5. Queue to be updated
      6. Output to an output file that contains all the processes of the program
2. **Decomposition Diagram**

|  |  |  |
| --- | --- | --- |
| **Main** | | |
| **Input** | **Process** | **Output** |
| Random times | Take a job out | Current time |
| Jobs that come in | Take a job in | All jobs currently in queue |
| Arrival time | Updates Queue | Jobs taken out |
| Processing time | Perform checks | Jobs taken in |

1. **Test Strategy**
   1. **Valid Data**
   2. **Invalid Data**
2. **Test Plan Version 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1 | Arrival time of job A is between 4 and 6 |  |  |  |  |
| Valid | 2 | Arrival time of job B is between 9 and 11 |  |  |  |  |
| Valid | 3 | Arrival time of job C is between 24 and 26 |  |  |  |  |
| Valid | 4 | Arrival time of job D is between 25 to 35 |  |  |  |  |
| Valid | 5 | Processing time of job A is between 1 to 5 |  |  |  |  |
| Valid | 6 | Processing time of job B is between 6 to 10 |  |  |  |  |
| Valid | 7 | Processing time of job C is between 11 to 15 |  |  |  |  |
| Valid | 8 | Processing time of job D is between 8 to 12 |  |  |  |  |
|  |  |  |  |  |  |  |

1. **Initial Algorithm**
   1. An array to be for priority queues. This array will be used as heap operations
   2. Initialize a counter variable to be 0 to increment and count how much time a job stays to be processed
   3. Make a struct that contains processing time value (int) and arriving time value (int)
   4. Make a struct of CPU that contains count variable for counting processing time, current job being processed (char)
   5. Make 4 arrays of type <struct> to store all the random values for arrival times and processing times for each job
      1. For arriving time values
         1. Array for A, index 1 starts with having a random number 5 +/- 1. Then adding up from this number and the random number range, prefill the whole array
         2. Array for B, index 1 starts with having a random number 10 +/- 1. Then adding up from this number and the random number range, prefill the whole array
         3. Array for C, index 1 starts with having a random number 25 +/- 1. Then adding up from this number and the random number range, prefill the whole array.
         4. Array for D, index 1, starts with having a random number 30 +/- 5. Then adding up from this number and the random number range, prefill the whole array.
      2. For processing time values
         1. Prefill array for A with random processing times of 3 +/- 2
         2. Prefill array for B with random processing times of 8 +/- 2
         3. Prefill array for C with random processing times of 13 +/- 2
         4. Prefill array for D with random processing times of 10 +/- 2
   6. Main loop that loops for loops for 1000 times
   7. For the first 500 time, do not print out any messages to screen or the screen
   8. If queue’s highest priority’s arriving time value equals current time while the CPU’s current job is null
      1. Assign the highest priority job in the queue to CPU’s current job
      2. Rearrange the queue in terms of priority
      3. increment the processing time counter by 1
      4. Print out this <job> is being processed
   9. If queue is empty while the CPU’s current job is NULL
      1. Print out NO JOB BEING PROCESSED
   10. If CPU’s current job’s processing time equals the counter while CPU is not null
       1. Check if queue is empty
          1. If queue is not empty, then assign the highest priority job in the queue to CPU’s current job, re-organize the queue in terms of priority
          2. If queue is empty, then set CPU’s current job to NULL
   11. If current time matches one of the arriving time values in the array for any job while CPU is not NULL
       1. Put it in the queue and reorganize it in terms of priority
2. **Test Plan Version 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1 | Arrival time of job A is between 4 and 6 | 4 + rand()%3 | 5 +/- 1 |  |  |
| Valid | 2 | Arrival time of job B is between 9 and 11 | 9 + rand()%3 | 10 +/- 1 |  |  |
| Valid | 3 | Arrival time of job C is between 24 and 26 | 24 + rand()%3 | 25 +/- 1 |  |  |
| Valid | 4 | Arrival time of job D is between 25 to 35 | 25 + rand()%11 | 30 +/- 5 |  |  |
| Valid | 5 | Processing time of job A is between 1 to 5 | 1 + rand()%5 | 3 +/- 2 |  |  |
| Valid | 6 | Processing time of job B is between 6 to 10 | 6 + rand()%5 | 8 +/- 2 |  |  |
| Valid | 7 | Processing time of job C is between 11 to 15 | 11 + rand()%5 | 13 +/- 2 |  |  |
| Valid | 8 | Processing time of job D is between 8 to 12 | 8 + rand()%5 | 10 +/- 2 |  |  |
| Invalid | 1 | Random value generated is negative | -4 for arrival time for job A | “Error in job arrival time value” |  |  |

1. **Code**

#include <iostream>

#include <string>

#include <fstream>

#include <time.h>

#include <vector>

using namespace std;

struct queue {

char job;

int arrivalTime;

int processTime;

};

struct CPU {

int timeCount;

char job;

};

void addInQueue(int time, queue jobs[], int &indexQueue, queue A[], int &indexA, queue B[], int &indexB, queue C[], int &indexC, queue D[], int &indexD);

void takeOutQueue(queue jobs[], int &indexQueue);

void preFillJobTimes(queue A[], queue B[], queue C[], queue D[], int spaceArriveA, int spaceArriveB, int spaceArriveC, int spaceArriveD);

void organizeQueue(queue jobs[], int size);

int main()

{

int counter = 0;

queue jobs[10000];

for (int i = 0; i < 10000; i++)

{

jobs[i].arrivalTime = 0;

jobs[i].job = 'E';

jobs[i].processTime = 0;

}

CPU cpu;

queue A[10000], B[10000], C[10000], D[10000];

int indexA = 0, indexB = 0, indexC = 0, indexD = 0;

int indexQueue = 0;

preFillJobTimes(A, B, C, D, 4, 9, 24, 25);

/\*jobs[0].job = 'A';

jobs[0].arrivalTime = 2;

jobs[0].processTime = 4;

jobs[1].job = 'B';

jobs[1].arrivalTime = 4;

jobs[1].processTime = 4;\*/

cpu.job = 'E';

cpu.timeCount = 0;

for (int time = 1; time < 10001; time++)

{

addInQueue(time, jobs, indexQueue, A, indexA, B, indexB, C, indexC, D, indexD);

/\*addInQueue(time, jobs, indexQueue, A, indexA, B, indexB, C, indexC, D, indexD);\*/

cout << time << "). ";

if (cpu.job == 'E') { //If CPU is empty

if (jobs[0].job != 'E' && jobs[0].arrivalTime == time) //If there is a job coming in from queue

{

cpu.job = jobs[0].job; //Assign cpu that job

cpu.timeCount++; //Iterate counter

}

}

else if (cpu.job != 'E') //If CPU is full

{

if (jobs[0].processTime == cpu.timeCount) //If the current job is going out

{

//int j;

//for (int i = 0; jobs[i+1].job != NULL; i++) //Move every job in the queue to the left

//{

// jobs[i].arrivalTime = jobs[i + 1].arrivalTime;

// jobs[i].job - jobs[i + 1].job;

// jobs[i].processTime = jobs[i + 1].processTime;

// j = i;

//}

takeOutQueue(jobs, indexQueue);

//jobs[j].arrivalTime = 0;

//jobs[j].job = NULL;

//jobs[j].processTime = 0;

//jobs[9999].arrivalTime = 0;

//jobs[9999].job = NULL;

//jobs[9999].processTime = 0;

cpu.timeCount = 0; //reset counter

cpu.job = 'E'; //reset current job

if (jobs[0].job != 'E' && jobs[0].arrivalTime <= time) //If there IS a job coming in from the queue

{

cout << "Job coming in: " << jobs[0].job;

cpu.job = jobs[0].job; //Assign cpu that job

cpu.timeCount++; //Iterate counter

}

}

else { //If the current job is not going out

cpu.timeCount++; //Iterate counter

//Reorganize the queue if there is a job coming in

addInQueue(time, jobs, indexQueue, A, indexA, B, indexB, C, indexC, D, indexD);

organizeQueue(jobs, indexQueue + 1);

}

}

if (cpu.job != 'E')

{

cout << cpu.job;

cout << ". Total jobs in queue: " << indexQueue + 1 << endl;

}

else {

cout << 'E' << endl;

}

}

//A.arrivalTime = 4 + rand()%3;

//A.processTime = 1 + rand()%5;

//A.job = 'A';

//B.arrivalTime = 9 + rand()%3;

//B.processTime = 6 + rand()%5;

//C.arrivalTime = 24 + rand()%3;

//C.processTime = 11 + rand()%5;

//D.arrivalTime = 25 + rand()%11;

//D.processTime = 8 + rand()%5;

return 0;

}

void addInQueue(int time, queue jobs[], int &indexQueue, queue A[], int &indexA, queue B[], int &indexB, queue C[], int &indexC, queue D[], int &indexD)

{

if (D[indexD].arrivalTime == time) //If job D is coming in

{

for (int i = indexQueue; i > -1; i--)

{

jobs[i + 1].processTime = jobs[i].processTime;

jobs[i + 1].job = jobs[i].processTime;

jobs[i + 1].arrivalTime = jobs[i].arrivalTime;

}

jobs[0].arrivalTime = D[indexD].arrivalTime;

jobs[0].job = 'D';

jobs[0].processTime = D[indexD].processTime;

indexD++;

indexQueue++;

}

else //If other jobs are coming in

{

if (A[indexA].arrivalTime) //If job A is coming in

{

jobs[indexQueue + 1].arrivalTime = A[indexA].arrivalTime;

jobs[indexQueue + 1].job = 'A';

jobs[indexQueue + 1].processTime = A[indexA].processTime;

indexA++;

indexQueue++;

organizeQueue(jobs, indexQueue + 1);

}

if (B[indexB].arrivalTime == time) //If job B is coming in

{

jobs[indexQueue + 1].arrivalTime = B[indexB].arrivalTime;

jobs[indexQueue + 1].job = 'B';

jobs[indexQueue + 1].processTime = B[indexB].processTime;

indexB++;

indexQueue++;

organizeQueue(jobs, indexQueue + 1);

}

if (C[indexC].arrivalTime == time) //If job C is coming in

{

jobs[indexQueue + 1].arrivalTime = C[indexC].arrivalTime;

jobs[indexQueue + 1].job = 'C';

jobs[indexQueue + 1].processTime = C[indexC].processTime;

indexC++;

indexQueue++;

organizeQueue(jobs, indexQueue + 1);

}

}

}

void takeOutQueue(queue jobs[], int &indexQueue)

{

for (int i = 0; i < indexQueue + 1; i++)

{

jobs[i].arrivalTime = jobs[i + 1].arrivalTime;

jobs[i].job = jobs[i + 1].job;

jobs[i].processTime = jobs[i + 1].processTime;

}

jobs[indexQueue].arrivalTime = 0;

jobs[indexQueue].job = 'E';

jobs[indexQueue].processTime = 0;

indexQueue--;

}

void organizeQueue(queue jobs[], int size)

{

int j;

for (int i = 0; i < size - 1; i++)

{

if (jobs[i].job != 'D') {

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

{

if (jobs[i].processTime > jobs[j].processTime)

{

int tempProcess = jobs[i].processTime;

char tempJob = jobs[i].job;

int tempArrive = jobs[i].arrivalTime;

jobs[i].processTime = jobs[j].processTime;

jobs[i].job = jobs[j].job;

jobs[i].arrivalTime = jobs[j].arrivalTime;

jobs[j].processTime = tempProcess;

jobs[j].job = tempJob;

jobs[j].arrivalTime = tempArrive;

}

}

}

}

}

void preFillJobTimes(queue A[], queue B[], queue C[], queue D[], int spaceArriveA, int spaceArriveB, int spaceArriveC, int spaceArriveD)

{

for (int i = 0; i < 10000; i++)

{

A[i].arrivalTime = 0;

A[i].job = 'E';

A[i].processTime = 0;

B[i].arrivalTime = 0;

B[i].job = 'E';

B[i].processTime = 0;

C[i].arrivalTime = 0;

C[i].job = 'E';

C[i].processTime = 0;

D[i].arrivalTime = 0;

D[i].job = 'E';

D[i].processTime = 0;

if (spaceArriveA + 6 < 10000)

{

A[i].arrivalTime = spaceArriveA + rand() % 3;

spaceArriveA = A[i].arrivalTime + (4 + rand() % 3);

A[i].processTime = 1 + rand() % 5;

//cout << "Current time: " << i << " Arrival time: " << A[i].arrivalTime << " Time: " << spaceArriveA - A[i].arrivalTime << " Process Time: " << A[i].processTime << endl;

}

if (spaceArriveB + 11 < 10000)

{

B[i].arrivalTime = spaceArriveB + rand() % 3;

spaceArriveB = B[i].arrivalTime + (9 + rand() % 3);

B[i].processTime = 6 + rand() % 5;

//cout << "Current time: " << i << " Arrival time: " << B[i].arrivalTime << " Time: " << spaceArriveB - B[i].arrivalTime << " Process Time: " << B[i].processTime << endl;

}

if (spaceArriveC + 26 < 10000)

{

C[i].arrivalTime = spaceArriveC + rand() % 3;

spaceArriveC = C[i].arrivalTime + (24 + rand() % 3);

C[i].processTime = 11 + rand() % 5;

//cout << "Current time: " << i << " Arrival time: " << C[i].arrivalTime << " Time: " << spaceArriveC - C[i].arrivalTime << " Process Time: " << C[i].processTime << endl;

}

if (spaceArriveD + 35 < 10000)

{

D[i].arrivalTime = spaceArriveD + rand() % 11;

spaceArriveD = D[i].arrivalTime + (25 + rand() % 11);

D[i].processTime = 8 + rand() % 5;

//cout << "Current time: " << i << " Arrival time: " << D[i].arrivalTime << " Time: " << spaceArriveD - D[i].arrivalTime << " Process Time: " << D[i].processTime << endl;

}

//else {

// cout << "Current time: " << i << " Arrival time: N/A Time: N/A Process Time: N/A" << endl;

//}

}

}

1. **Updated Algorithm**
   1. Make a struct named *queue* that stores char job, int arrival time, and int process time
   2. Make a struct names *CPU* that stores int time count, and char job
   3. In main()
      1. Set a counter variable to 0
      2. Make an array instance of *queue* struct and name is *jobs* that stores a total of 10000 set of struct values
      3. Using for loop going from 0 to 10000, Set the array’s each index arrival time to 0, job to *E*, and process time to 0
      4. Make an instance of *CPU* struct
      5. Make four array instances of *queue* struct
         1. One for job A that stores 10000 set of struct values
         2. One for job B that stores 10000 set of struct values
         3. One for job C that stores 10000 set of struct values
         4. One for job D that stores 10000 set of struct values
      6. Initialize 4 different counters for each type of job to keep track of the last index used to get the job and initialize all of them to 0
      7. Call the function to prefill job times
      8. Set cpu’s job to *E* and time count to 0
      9. Using for loop starting with time = 1 and go until time = 10000
         1. Call the function to add values in the queue
         2. Print out the time
         3. If cpu is empty
            1. If there is a job coming in from the queue

Assign cpu that job

Iterate counter

* + - 1. Else if cpu is full
         1. If the current job is going out

Call the function to take out the job from queue

Set cpu job to *E* and time count to 0

If there is a job coming in the queue

Print out the job name and iterate counter

* + - * 1. Else if there job is not going out

Iterate counter

Call the function to add job to the queue

Call the function to organize the queue in terms of priority

* + - 1. If current job is not *E* print the job name
      2. Else print *E*
  1. In *addInQueue()* function
     1. If job D is coming in
        1. Using for loop, starting from last index until 0
        2. Most everything to the right
     2. Set the first index values to be the values of job D
     3. If job A is coming in
        1. Put it the end of queue
     4. Do the same for each of the other jobs
     5. Organize the queue by calling the function
  2. In *takeOutQueue*() function
     1. Using for loop starting from 0 until last index
        1. Move everything to the left
     2. Set the last index values to *E* and default times
  3. In *organizeQueue()* function
     1. Using a for loop starting form 0 until 2nd to the last index
        1. If (i + 1 th index is not *D*)
           1. Using a for loop starting from i + 1 until size
           2. If ith index processor time is greater than its next index

Make temporary variable and set the ith index values to these variables

Set next index’s variabels to ith variables

Set the temporary variables to the next index’s variables

* 1. In *preFillJobTimes()* function
     + 1. Prefill array for A with random processing times of 3 +/- 2
       2. Prefill array for B with random processing times of 8 +/- 2
       3. Prefill array for C with random processing times of 13 +/- 2
       4. Prefill array for D with random processing times of 10 +/- 2

1. **Test Plan Version 3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid | 1 | Arrival time of job A is between 4 and 6 | 4 + rand()%3 | 5 +/- 1 |  | Pass |
| Valid | 2 | Arrival time of job B is between 9 and 11 | 9 + rand()%3 | 10 +/- 1 |  | Pass |
| Valid | 3 | Arrival time of job C is between 24 and 26 | 24 + rand()%3 | 25 +/- 1 |  | Pass |
| Valid | 4 | Arrival time of job D is between 25 to 35 | 25 + rand()%11 | 30 +/- 5 |  | Pass |
| Valid | 5 | Processing time of job A is between 1 to 5 | 1 + rand()%5 | 3 +/- 2 |  | Pass |
| Valid | 6 | Processing time of job B is between 6 to 10 | 6 + rand()%5 | 8 +/- 2 |  | Pass |
| Valid | 7 | Processing time of job C is between 11 to 15 | 11 + rand()%5 | 13 +/- 2 |  | Pass |
| Valid | 8 | Processing time of job D is between 8 to 12 | 8 + rand()%5 | 10 +/- 2 |  | Pass |
| Invalid | 1 | Random value generated is negative | -4 for arrival time for job A | “Error in job arrival time value” |  | Fail |

1. **Screenshots**
2. **Status**

Program does not work properly with the assumptions and logic in mind. Queues always keeps increasing. Only one job comes in and goes out properly. Program is inefficient to determine how many CPUs to be used