Andrea Abed

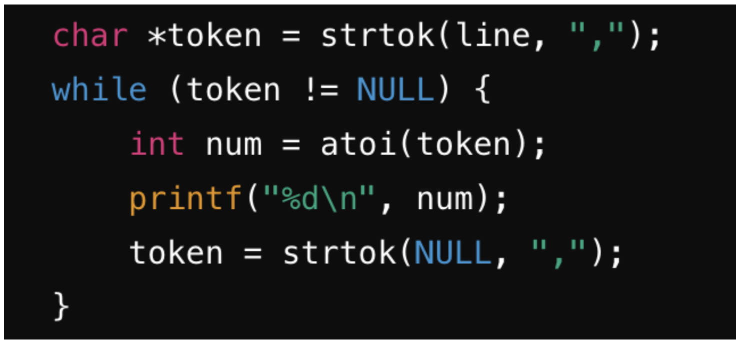
CS 480 - Professor Ramamoorthy

28 June 2024

Lab 3 Report - Scheduling Algorithms

I was able to complete this lab almost entirely. The only problem I noticed was when I tested the test3.txt file, for the shortest time remaining gantt chart, it did not print process 6 from time 90 to 121 as it should have, but it only printed from 90 to 100. However, the rest of the tests for all the algorithms on each of the test files worked as expected. The FCFS and Priority scheduling were the easiest algorithms to complete since they were nonpreemptive so it was more straightforward. The Shortest remaining time and Round Robin algorithms were more difficult especially when it came to printing the gantt chart. For those two algorithms, I decided on making an array named gantt_chart, where each element was a structure called "entry" that held 3 pieces of information: the process id, the time it started, and the time it ended. When it came to printing, it would accumulate all the entries that were adjacent to each other and who had the same process id, and would print the start time of the first entry for the process id and the last entry for the process id. Overall, this assignment helped me strengthen my understanding of the different scheduling algorithms, and made me more comfortable with file i/o, using structures, using pointers, dynamically allocating memory, etc. **Below are the sources I used:**

- For file i/o: https://www.geeksforgeeks.org/basics-file-handling-c/, https://www.geeksforgeeks.org/c-program-to-read-contents-of-whole-file/
- Help from chat gpt for getting each integer from each line/string in the file:



```c
char *token = strtok(line, ",");
while (token != NULL) {
    int num = atoi(token);
    printf("%d\n", num);
    token = strtok(NULL, ",");
}
```

- For dynamically allocating memory for a 2D array (also referenced project from CS420): https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/
- Referenced Lab 1 for idea of using a structure to hold information of a process and storing it in a table
- Referenced Lab 1 for creating a ready list that is a pointer to a linked list that has enqueue and dequeue functions that were helpful with the preemptive algorithms
- For round robin check_for_new_arrivals method, inspo from geeks for geeks:

https://www.geeksforgeeks.org/round-robin-scheduling-with-different-arrival-times/?ref=ml_lbp

Source Code:

```c
// gcc -g -Wall -o main  main.c
// ./main test1.txt

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


//structure for processes
typedef struct {
   int ID;
   int arrival;
   int burst;
   int priority;
   int time_quantum;
   int waiting;
   int turnaround;
   int remaining_burst_time;
} Process;


// structure for the queue/ready list
struct node {
   int process_id;
   struct node* next_process;
};

// initialize the ready list as null
struct node* ready_list = NULL;


// add a process to the end of the ready list
void enqueue(int id) {
   struct node* new_node = (struct node*) malloc(sizeof(struct node));
   new_node->process_id = id;
   new_node->next_process = NULL;

   // if nothing in ready list, have ready list point directly to the new process
being enqueued
   if (ready_list == NULL) {
       ready_list = new_node;
   }

   // otherwise, traverse the list to the end and add the new process
   else {
       struct node* last = ready_list;
       while (last->next_process != NULL) {
           last = last->next_process;
       }
       last->next_process = new_node;
   }
}

// remove a process from the beginning of the ready list
void dequeue() {
   if (ready_list != NULL) {
       struct node* temp = ready_list;
       ready_list = ready_list->next_process;
       free(temp);
   }
}


// structure for the gantt chart
typedef struct entry {
 int process_id;
 int start_time;
 int end_time;
} entry;


// make an array of entries, for printing gantt chart later
entry gantt_chart[100];


// function for adding an entry to the gantt chart
void add(int id, int start, int end){
   int i;
 for (i = 0; i < 100; i++){
   //find the first empty entry in the gantt chart array to update
   if (gantt_chart[i].process_id == -1){
     gantt_chart[i].process_id = id;
```

```
     gantt_chart[i].start_time = start;
     gantt_chart[i].end_time = end;
     break;
   }
 }
}

// printing the gantt chart
void print_gantt_chart() {
   int start_time = -1;
   int end_time = -1;
   int current_process_id = -1;

   // go through each entry
   int i;
   for (i = 0; i < 100; i++) {

       // break for loop once the end of the gantt chart is reached
       if (gantt_chart[i].process_id == -1) {
           break;
       }

       // for the first entry
       if (current_process_id == -1) {
           // initialize the current_process_id
           current_process_id = gantt_chart[i].process_id;
           // get the start time
           start_time = gantt_chart[i].start_time;
           // and what is assumed to be the end time at this point
           end_time = gantt_chart[i].end_time;
       }
       // if adjacent entries have the same process id
       else if (gantt_chart[i].process_id == current_process_id) {
           // update the end time
           end_time = gantt_chart[i].end_time;
       }

       else {
           // if the current entry has a different process id than the previous entry,
print the gantt chart of the previous entry
           if (current_process_id == 0) {
               // if it was 0, it was idle
```

```
                printf("[%d]-- IDLE --[%d]\n", start_time, end_time);
            } else {
                // otherwise print the id
                printf("[%d]-- %d --[%d]\n", start_time, current_process_id, end_time);
            }
            // update the current_process_id as the new process id
            current_process_id = gantt_chart[i].process_id;
            start_time = gantt_chart[i].start_time;
            end_time = gantt_chart[i].end_time;
        }
    }

    // once id = -1, loop breaks, print the previous entry
    if (current_process_id != -1) {
        if (current_process_id == 0) {
            printf("[%d]-- IDLE --[%d]\n", start_time, end_time);
        } else {
            printf("[%d]-- %d --[%d]\n", start_time, current_process_id, end_time);
        }
    }
}


// helper function for round robin
void check_for_new_arrivals(Process* table, int rows, int current_time, int* in_list,
int* finished) {
    int i, j;
    for (i = 0; i < rows; i++) {
        int earliest_index = -1;
        //for each process
        for (j = 0; j < rows; j++) {
            int process_num = table[j].ID;
            // if a new process has arrived, is not currently in the queue, and is not
finished yet
            if (table[j].arrival <= current_time && in_list[process_num] == 0 &&
finished[process_num] == 0) {
                // if it arrived first or if it arrived at the same time as another
process but has a smaller id, select
                if (earliest_index == -1 || table[j].arrival <
table[earliest_index].arrival || (table[j].arrival == table[earliest_index].arrival &&
table[j].ID < table[earliest_index].ID)) {
                    earliest_index = j;
                }
            }
        }
        // enqueue the process that was selected
        if (earliest_index != -1) {
            int process_num = table[earliest_index].ID;
            in_list[process_num] = 1;
            enqueue(process_num);
        }
    }
}



void RR(Process *table, int rows, int time_quantum) {
   printf("\n\n\nfrom inside Round Robin\n\n\n");

   // initialize the gantt chart array
   int i;
   for (i = 0; i < 100; i++){
     gantt_chart[i].process_id = -1;
     gantt_chart[i].start_time = -1;
     gantt_chart[i].end_time = -1;
   }

   // initialize number of completed, current time, in_list (keeps track of which
process are in the queue), and finished (keeps track of which process are finished)
   int completed = 0;
   int current_time = 0;

   int in_list[rows + 1];
   int finished[rows + 1];

   for (i = 0; i <= rows; i++) {
       in_list[i] = 0;
       finished[i] = 0;
   }

   // initialize remaining burst time for each process
   for (i = 0; i < rows; i++) {
       table[i].remaining_burst_time = table[i].burst;
   }
```

```
    // initial check for new arrivals and enqueue them
    check_for_new_arrivals(table, rows, current_time, in_list, finished);

    // while all process are not finished
    while (completed != rows) {

        // while there is no process in the queue
        while (ready_list == NULL) {
            // cpu is idle
            for (i = 0; i < 100; i++) {
                // add entry to gantt chart
                if (gantt_chart[i].process_id == -1) {
                    add(0, current_time, current_time + 1);
                    break;
                }
            }
            current_time++;
            //increment current time by one, check again for new arrivals with the new
time, when one is found, while loop breaks
            check_for_new_arrivals(table, rows, current_time, in_list, finished);
        }

        //once arrival is found
        struct node *current_process = ready_list;
        int current_process_id = current_process->process_id;

        //dequeue from the ready list
        dequeue();

        // find the index in which the process resides in the table
        int current_process_index = -1;
        for (i = 0; i < rows; i++) {
            if (table[i].ID == current_process_id) {
                current_process_index = i;
                break;
            }
        }

        //track start time for gantt chart
        int start_time = current_time;
        // if remaining burst <= time quantum, the process is completed
        if (table[current_process_index].remaining_burst_time <= time_quantum) {
            completed++;
            // update current time to however much time was left
            current_time += table[current_process_index].remaining_burst_time;

            // add to gantt chart for printing
            add(current_process_id, start_time, current_time);

            // store waiting time and turnaround time
            table[current_process_index].waiting = current_time - 
table[current_process_index].arrival - table[current_process_index].burst;
            table[current_process_index].turnaround = 
table[current_process_index].waiting + table[current_process_index].burst;

            // make sure waiting time is not negative
            if (table[current_process_index].waiting < 0) {
                table[current_process_index].waiting = 0;
            }

            // update remaining time, and mark as finished
            table[current_process_index].remaining_burst_time = 0;
            finished[current_process_id] = 1;

            // check for new arrivals to enqueue to readylist
            for (i = 1; i <= rows; i++) {
                if (in_list[i] == 0) {
                    check_for_new_arrivals(table, rows, current_time, in_list, 
finished);
                    break;
                }
            }

        }

            // if remaining burst > time quantum, the process is not completed
        else {
            // update the remaining burst time to reflect the time gone by
            table[current_process_index].remaining_burst_time -= time_quantum;
            // update the current time
            current_time += time_quantum;
            // add entry to gantt chart
            add(current_process_id, start_time, current_time);

            // check for new arrivals to enqueue to readylist
            for (i = 1; i <= rows; i++) {
                if (in_list[i] == 0) {
                    check_for_new_arrivals(table, rows, current_time, in_list,
finished);
                    break;
                }
            }

            // enqueue the current process after enqueueing the new arrivals
            enqueue(current_process_id);
        }
    }

    // print Gantt chart after all processes have completed
    print_gantt_chart();


    printf("\n\n-------------- RR ----------------\n");
    printf("Process ID | Waiting Time | Turnaround Time\n");

    double avg_waiting = 0;
    double avg_turnaround = 0;
    double throughput = 0;

    for (i = 0; i < rows; i++) {
        printf("%d\t\t%d\t\t%d\n", table[i].ID, table[i].waiting, table[i].turnaround);
        avg_waiting += table[i].waiting;
        avg_turnaround += table[i].turnaround;
    }
    printf("--------------------------------------\n");

    avg_waiting /= rows;
    avg_turnaround /= rows;
    throughput = (double)rows / current_time;

    printf("\n\nAverage Waiting Time: %.1f\n", avg_waiting);
    printf("Average Turnaround Time: %.1f\n", avg_turnaround);
    printf("Throughput: %.10f\n", throughput);
```

```
}




//helper for priority scheduling
int find_next_arrival(Process* table, int rows, int* checked){

   //finds the process with the next minimum arrival time, and returns the actual
minimum time
   int min_arrival = __INT_MAX__;
   int i;
   for (i = 0; i < rows; i++){
       if (checked[table[i].ID] == 0 && table[i].arrival < min_arrival){
           min_arrival = table[i].arrival;
       }
   }
   return min_arrival;
}

//helper for priority scheduling
int select_process(Process* table, int rows, int* checked, int current_time){
   int min_priority = __INT_MAX__;
   int min_priority_process = -1;
   int selected_index = -1;
   int i;
   for (i = 0; i < rows; i++){
       // if process has not been selected yet, and the process has already arrived by
the current time
     if (checked[table[i].ID] == 0 && table[i].arrival <= current_time){
         // if the process has a lower priority than the current minimum priority, or
if it is the same as the minimum priority but has a lower ID, then update the minimum
priority and minimum priority process
         if (table[i].priority < min_priority || (table[i].priority == min_priority &&
table[i].ID < min_priority_process)){
             min_priority = table[i].priority;
             min_priority_process = table[i].ID;
             // return the index of the table corresponding to the selected processs
             selected_index = i;
         }
     }
```

```
 }
   // if a process has been selected, then update the checked array and return the
index of the selected process
 if (min_priority_process != -1) {
     checked[min_priority_process] = 1;
 }
 return selected_index;
}



void PS(Process* table, int rows){
   printf("\n\nfrom inside Priority Scheduling\n");

   // checked array to keep track of which processes have already been selected (index
corresponds to the process id)
   int checked[rows+1];
   int i;
   for (i = 0; i < rows+1; i++){
       checked[i] = 0;
   }

   int current_time = 0;

   printf("\nGantt Chart is:\n");

   // loop until all processes have been selected
   for (i = 0; i < rows; i++){
       // select process based on who has arrived already, who has the lowest (aka
most) priority, and if there is a tie, who has the smaller process id
       int selected_process = select_process(table, rows, checked, current_time);

       // if no process has arrived yet (begin idle from this point)
       if (selected_process == -1) {

           // call another function to find the actual time that the next process
arrives (earlieset arrival time)
           int next_arrival_time = find_next_arrival(table, rows, checked);

           // if there exists a next arrival time (idle from the current time until
the next arrival time)
           if (next_arrival_time != __INT_MAX__) {
               printf("[ %d ]-- IDLE --[ %d ]\n", current_time, next_arrival_time);
               current_time = next_arrival_time;

               // decrement i, and go back to the top of the loop, to retry with the
new current time
               i--;
               continue;
           }
       }

       // otherwise, if there is a process that has arrived under the desired
conditions
       // hold its info in current process which points to its location in the table
       Process *current_process = &table[selected_process];

       // like FCFS... redundant?
       // if the current time is less than the process who has arrived, then CPU is
idle until the point that process has arrived
       if (current_time < current_process->arrival) {
           printf("[ %d ]-- IDLE --[ %d ]\n", current_time, current_process->arrival);
           current_time = current_process->arrival;
       }

       // otherwise the process uses the CPU without being interrupted
       printf("[ %d ]-- %d --[ %d ]\n", current_time, current_process->ID,
current_time + current_process->burst);
       current_time += current_process->burst;

       //get waiting and turnaround for that particular process
       current_process->turnaround = current_time - current_process->arrival;
       current_process->waiting = current_process->turnaround -
current_process->burst;

   }

   //printf("current time is %d\n", current_time);

   printf("\n\n--------------- PS ----------------\n");
   printf("Process ID | Waiting Time | Turnaround Time\n");
   //accumulate waiting and turnaround time of all the processes and then divide by
number of processes
   double avg_waiting = 0;
   double avg_turnaround = 0;
   double throughput = 0;

   for (i = 0; i < rows; i++){
       printf("%d\t\t%d\t\t%d\n", table[i].ID, table[i].waiting, table[i].turnaround);
       avg_waiting += table[i].waiting;
       avg_turnaround += table[i].turnaround;
   }
   printf("-------------------------------------\n");

   double num_rows = rows;

   avg_waiting = avg_waiting / rows;
   avg_turnaround = avg_turnaround / rows;

   //throughput is the number of jobs completed in what amount of time
   throughput = num_rows / current_time;

   printf("\n\nAverage Waiting Time: %.2f\n", avg_waiting);
   printf("Average Turnaround Time: %.2f\n", avg_turnaround);
   printf("Throughput: %.12f\n", throughput);
}






void SRT(Process* table, int rows) {
   printf("\n\nfrom inside Shortest Time Remaining\n");

   // initialize gantt chart array
   int i;
   for (i = 0; i < 100; i++){
     gantt_chart[i].process_id = -1;
     gantt_chart[i].start_time = -1;
     gantt_chart[i].end_time = -1;
   }

   // initialize current time, completed, and finished array
   int current_time = 0;
   int completed = 0;

   int finished[rows];
   for (i = 0; i < rows; i++) {
       finished[i] = 0;
       table[i].remaining_burst_time = table[i].burst;
   }

   // while there are still processes to be completed
   while (completed != rows) {

       //assume no process is found yet
       int min_burst_index = -1;
       int min_remaining_burst_time = __INT_MAX__;

       //for each process in table
       for (i = 0; i < rows; i++) {
           //if process has already arrived into the system and still has more
remaining burst time
           if (table[i].arrival <= current_time && finished[i] == 0) {
               // if the remaining time is less than what was previously thought to be
the minimum
               if (table[i].remaining_burst_time < min_remaining_burst_time) {
                   //update the minimum burst time and index it corresponds to
                   min_remaining_burst_time = table[i].remaining_burst_time;
                   min_burst_index = i;
               // if there are multiple processes that have the same minimum remaining
time, select the one with the lowest ID
               if (table[i].remaining_burst_time == min_remaining_burst_time &&
table[i].ID < table[min_burst_index].ID) {
                       min_burst_index = i;
                   }
               }
           }
       }



       // if a process exists in the system and is selected as the one with the
shortest remaining time...
       if (min_burst_index != -1) {
           //update its remaining burst time to account for passing time
           table[min_burst_index].remaining_burst_time--;
           add(table[min_burst_index].ID, current_time, current_time + 1);
           current_time++;

           // if the current process utilizing the CPU has completed by this time
           if (table[min_burst_index].remaining_burst_time == 0) {

               // save the turnaround time and waiting time of the process
               table[min_burst_index].turnaround = current_time -
table[min_burst_index].arrival;
               table[min_burst_index].waiting = table[min_burst_index].turnaround -
table[min_burst_index].burst;

               // mark the process as finished
               finished[min_burst_index] = 1;

               // increment the number of completed processes
               completed++;
           }
       }

       // otherwise, if no process is available to run...
       else {
           add(0, current_time, current_time + 1);
           current_time++;
       }
   }

   double avg_waiting = 0;
   double avg_turnaround = 0;

   printf("\n\nGantt Chart is:\n\n");
   print_gantt_chart();

   printf("\n\n--------------- SRT ----------------\n");
   printf("Process ID | Waiting Time | Turnaround Time\n");
   for (i = 0; i < rows; i++) {
       avg_waiting += table[i].waiting;
       avg_turnaround += table[i].turnaround;
       printf("%d\t\t%d\t\t%d\n", table[i].ID, table[i].waiting, table[i].turnaround);
   }
```

```c
   printf("-------------------------------------\n");

   avg_waiting /= rows;
   avg_turnaround /= rows;
   double throughput = (double) rows / current_time;

   printf("\n\nAverage Waiting Time: %.2f\n", avg_waiting);
   printf("Average Turnaround Time: %.2f\n", avg_turnaround);
   printf("Throughput: %.12f\n", throughput);
}











// helper for first come first serve
//returns process with earliest arrival and smallest process id
int get_min_process(Process* table, int rows, int* checked){

   int min_arrival = __INT_MAX__;
   int min_arrival_process = -1;
   // a to keep track of index of the minimum process in the table
   int a = -1;

   int i;
   for (i = 0; i < rows; i++){
       //if process has not been previously selected as the minimum
       //and the arrival time of the process is less than the current minimum arrival
time saved
       //or if it is equal to the minimum, but has a lower process id than the current
minimum process id
       //then update the minimum arrival time and corresponding process id
       if (checked[table[i].ID] == 0 && (table[i].arrival < min_arrival ||
(table[i].arrival == min_arrival && table[i].ID < min_arrival_process))){
           min_arrival = table[i].arrival;
           min_arrival_process = table[i].ID;
           a = i;
       }
```

```
    }

    // if minimum process was found, update its corresponding position in checked array
    if (min_arrival_process != -1) {
        checked[min_arrival_process] = 1;
    }

    return a;
}




void FCFS(Process* table, int rows){
    printf("\n\nfrom inside First Come First Serve\n");

    // checked array to keep track of which processes have already been selected (index
corresponds to the process id)
    int checked[rows+1];
    int i;
    for (i = 0; i < rows+1; i++){
        checked[i] = 0;
        //printf("checked[%d], is %d\n", i, checked[i]);
    }

    int current_time = 0;

    printf("\nGantt Chart is:\n");


    // for all processes
    for (i = 0; i < rows; i++){

        // get the process that arrives first (if tie, get the process with smallest
process id)
        // get its index in the table (different from the ID itself)
        int min_process_index = get_min_process(table, rows, checked);

        // have current process point to that process and its information in the table
        Process *current_process = &table[min_process_index];
        // if the current time is less than the process who has arrived, then CPU is
idle until the point that process has arrived
        if (current_time < current_process->arrival) {
                printf("[ %d ]-- IDLE --[ %d ]\n", current_time,
current_process->arrival);
                current_time = current_process->arrival;
        }

        //other wise, the process takes up the CPU for its burst time (not interrupted)
        printf("[ %d ]-- %d --[ %d ]\n", current_time, current_process->ID,
current_time + current_process->burst);
        // current time updates to account for the burst time of the process
        current_time += current_process->burst;


        // calculate turnaround time and waiting time for that process and store the
info
        current_process->turnaround = current_time - current_process->arrival;
        current_process->waiting = current_process->turnaround -
current_process->burst;
        }

    //printf("current time is %d\n", current_time);


    printf("\n\n--------------- FCFS ----------------\n");
    printf("Process ID | Waiting Time | Turnaround Time\n");


    double avg_waiting = 0;
    double avg_turnaround = 0;
    double throughput = 0;

    // for each process, accumulate the waiting time and turnaround time
    for (i = 0; i < rows; i++){
        printf("%d\t\t%d\t\t%d\n", table[i].ID, table[i].waiting, table[i].turnaround);
        avg_waiting += table[i].waiting;
        avg_turnaround += table[i].turnaround;
    }
    printf("--------------------------------------\n");
```

```
   // divide waiting and turnaround time by the number of processes to get the
average
   double num_rows = rows;
   avg_waiting = avg_waiting / rows;
   avg_turnaround = avg_turnaround / rows;
   throughput = num_rows / current_time;

   printf("\n\nAverage Waiting Time: %.2f\n", avg_waiting);
   printf("Average Turnaround Time: %.2f\n", avg_turnaround);
   printf("Throughput: %.12f\n", throughput);

}




int main(int argc, char* argv[]) {
   printf("Hello World\n");

   if (argc < 2) {
       printf("Too few arguments. Provide name of file.\n");
       return 1;
   }

   if (argc > 2) {
       printf("Too many arguments. Only provide name of file as a single
argument.\n");
       return 1;
   }

   //pointer to filename of data type character
   char* filename = argv[1];

   //fopen returns file pointer if opened successfully
   FILE *file = fopen(filename, "r");

   if (file == NULL) {
       printf("Unable to open file\n");
       return 1;
   }

   //line length
   char line[100];

   //number of columns always constant
   int columns = 5;
   int rows = 0;

   // obtain number of rows which will vary among text file inputs
   while (fgets(line, 100, file)){
     rows++;
   }

   printf("Number of rows: %d\n", rows);


   //rewind file pointer to the beginning of the file
   rewind(file);


   // creating 2D array to resemble text file
   int **info_2Darray = (int**)malloc(rows * sizeof(int**));

   // allocate memory for each row (dynamic because rows/# of process depend on the
input file)
   if (info_2Darray == NULL){
       printf("Unable to allocate memory for rows");
       return 1;
   }

   //allocate memory for the columns in each row
   int i;
   for (i = 0; i < rows; i++){
       info_2Darray[i] = (int*)malloc(columns * sizeof(int));
       if (info_2Darray[i] == NULL){
           printf("Unable to allocate memory for the columns in row %d\n", i);
           return 1;
       }
   }


   int row, column;

   // for each row
```

```c
    for (row = 0; row < rows; row++){
        // reset column to leftmost column when moving to next row
        column = 0;

        // read line from file
        // fgets(line of data type string that will store the line of data, maximum
characters that will be stored, file pointer)
        fgets(line, 100, file);


        // strtok is a function that takes a string and a delimiter and returns a
pointer to the first token in the string
        // in this case, token is each individual number in the line
        char *token = strtok(line, ",");

        while (token != NULL && column < columns) {

            // converts number in the string to an integer
            int num = atoi(token);

            // stores the integer in its corresponding position in 2D array
            info_2Darray[row][column] = num;

            printf("%d ", info_2Darray[row][column]);

            // moves to the next token in the string
            token = strtok(NULL, ",");

            // increments column to keep track within 2D array
            column++;
        }

      printf("\n");
    }

    // finished reading from the file, everything now contained in 2D array, close file
    fclose(file);


    // each row in the 2D array corresponds to: proccess id, arrival time, burst time,
priority, and time quantum
```

```c
    // create a table of process structures, each element in 1D array will hold the
information for that process
    Process TABLE[rows];

    // initialize values in table of processes
    for (i = 0; i < rows; i++){
        TABLE[i].ID = info_2Darray[i][0];
        TABLE[i].arrival = info_2Darray[i][1];
        TABLE[i].burst = info_2Darray[i][2];
        TABLE[i].priority = info_2Darray[i][3];
        TABLE[i].time_quantum = info_2Darray[i][4];
        TABLE[i].waiting = 0;
        TABLE[i].turnaround = 0;
        TABLE[i].remaining_burst_time = 0;
    }

    // call the scheduling algorithms
    FCFS(TABLE, rows);
    SRT(TABLE, rows);
    PS(TABLE, rows);
    RR(TABLE, rows, TABLE[0].time_quantum);



    // free the memory allocated for the 2D array
    for (i = 0; i < rows; i++) {
      free(info_2Darray[i]);
    }

    free(info_2Darray);




    return 0;
}
```