Updated:
20/04/2023

Operating Systems
Project C programming

Non-Preemptive Process Scheduling Algorithms

Deadline: 7 Mai 2023 @ 23:59

Student Code Ethics

Students are expected to maintain the highest standards of academic integrity. Work that is not of the student's own creation will receive no credit, and disciplinary actions can follow. Remember that you cannot give or receive unauthorized aid on any assignment, quiz, or exam. A student cannot use the ideas of another and declare it as his or her own.

You are allowed to share unit tests to complement the file `test_scheduling.c`.

Submission

Upload a file `scheduling.c` on OLE.

Objectives

Write standard C code that satisfies a specification. Implement basic algorithms of process scheduling. Write correct, clean, reusable, and factorized code. Use unit testing to verify the correctness of the solutions.

Evaluation criteria

The grade will reflect the correctness, elegance, and overall quality of the code in the file `scheduling.c`. Any academic misconduct will lead to a failing grade.

Further remarks

- This project description may evolve; specially to correct mistakes or ambiguities.
- Signal any possible ambiguity in this description.
- Allocate only the memory necessary (the size of scheduling problems is unbounded; only when the user tells us the problem, do we know its size).
- Free memory as soon as possible.
- Avoid code duplication (the logics of FCFS and SJF are very similar).
- Make your code modular using functions and procedures, but don't overdo it.
- Comment your code to help reading, but don't overdo it.
- Avoid the use of constants without `#define`.
- Check the correctness of your implementation; you are encouraged to augment and share your version of `test_scheduling.c`.

Instructions

Download `scheduling.h`, and `test_scheduling.c`.

Create a file `scheduling.c` and implement the functions specified in `scheduling.h`.

```
typedef struct {
  int num;
  int **table;
} sch_problem;

typedef struct {
  int num;
  int *order;
  float wait_average;
} sch_solution;

void sch_table_malloc(sch_problem *sch);
void sch_table_free  (sch_problem *sch);
sch_problem  * sch_get_scheduling_problem_instance();
sch_solution * sch_fcfs(sch_problem *sch);
sch_solution * sch_sjf (sch_problem *sch);
```

You can compile the tests with, e.g., (`scheduling.c` must not contain a `main()`)

```
gcc -o testsched test_scheduling.c scheduling.c
```

`sch_problem` is a scheduling problem instance, and holds a number `num` of jobs, listed in a `table`, where each row contains the job `ID`, the job `ARRIVAL` time, and the job `BURST` time.

`sch_solution` is a scheduling solution structure, and holds a number `num` of jobs, an array representing the `order` of scheduling, and a `wait_average` time.

More details about `sch_problem` and `sch_solution` are in `scheduling.h`.

The functions specifications are as follows:

```
/**
  Allocate memory for the table in the scheduling problem structure sch in
  parameter. It allocates the memory to hold a matrix of int sch->num x 3.
  Each row i of the table holds the information about one job:
          sch->table[i][ID]      : i+1
          sch->table[i][ARRIVAL] : arrival time
          sch->table[i][BURST]   : burst time

  @param sch the address of the scheduling problem;
       sch->num must already contain the number of jobs
       in the instance.
*/
void sch_table_malloc(sch_problem *sch);


/**
  Free the memory occupied by the table of the scheduling
  problem at sch.

  @param sch the address of the scheduling problem
```

```
*/
void sch_table_free(sch_problem *sch);


/**
  Get a scheduling problem instance from the user.

  @return the address of the scheduling problem structure
          containing the instance provided by the user
*/
sch_problem * sch_get_scheduling_problem_instance();


/**
   Compute the solution to a scheduling problem with First Come
   First Served scheduling. Break ties in favour of the job with
   lower index in sch->table.

   @param sch the address of the scheduling problem to solve

   @return the address of the computer scheduling solution
 */
sch_solution * sch_fcfs(sch_problem *sch);


/**
   Compute the solution to a scheduling problem with Shortest
   Job First scheduling. Break ties in favour of the job with
   lower index in sch->table.

   @param sch the address of the scheduling problem to solve

   @return the address of the computer scheduling solution
 */
sch_solution * sch_sjf(sch_problem *sch);
```

Although it is *not part of the specification and not requested*, the functions `fcfs` and `sjf` could also print some details of the solution to the user as it is building it.

Examples

The code

```
sch_problem  *sch = sch_get_scheduling_problem_instance();
sch_solution *sol_fcfs = sch_fcfs(sch);
sch_solution *sol_sjf = sch_sjf(sch);
```

in a `main` function could be used to obtain the following interaction with the user:

```
How many jobs in the scheduling problem instance? 4
Job 1, arrival time: 2
Job 1, burst time: 2
Job 2, arrival time: 3
Job 2, burst time: 1
Job 3, arrival time: 1
Job 3, burst time: 4
Job 4, arrival time: 9
Job 4, burst time: 2
********** FCFS
(  0) No job ready.
(  1) Running job 3, arrived at 1, with burst time 4
(  5) Running job 1, arrived at 2, with burst time 2
(  7) Running job 2, arrived at 3, with burst time 1
(  8) No job ready.
(  9) Running job 4, arrived at 9, with burst time 2
Average wait time 1.750000.
********** SJF
(  0) No job ready.
(  1) Running job 3, arrived at 1, with burst time 4
(  5) Running job 2, arrived at 3, with burst time 1
(  6) Running job 1, arrived at 2, with burst time 2
(  8) No job ready.
(  9) Running job 4, arrived at 9, with burst time 2
Average wait time 1.500000.
```

Another interaction with the user:

Finally, the unit tests in `test_scheduling.h` are more precise examples of the expected behaviour of the function `sch_fcfs` and `sch_sjf`.