Data Structures

BLG 223E

Project 2

Res. Assist. Ali Esad Uğur
ugura20@itu.edu.tr

Instructors:

Prof. Dr. Tolga Ovatman
Asst. Prof. Dr. Yusuf Hüseyin Şahin

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 29.11.2024

1. Process Management System

1.1. Task Description

Dear all,

In this assignment, you are asked to build a very simple process management system, using the information you learned in the lectures about stacks and queues. Process management is a key part of an operating system. It controls how processes are carried out, and how your computer runs by handling the active processes. This includes stopping processes, setting which processes should get more attention, and many more.

The OS is responsible for managing the start, stop, and scheduling of processes, which are programs running on the system. It uses a number of methods to prevent deadlocks, facilitate inter-process communication, and synchronize processes. Efficient resource allocation, conflict-free process execution, and optimal system performance are all guaranteed by competent process management. This essential component of an operating system enables the execution of numerous applications at once, enhancing system utilization and responsiveness. For more information, you can refer to a resource about operating systems basics, or simply wait for the operating systems course in your curriculum.

1.2. Necessary System Specifications

The specific system that you will build is not as complicated as a regular process manager. You will only implement how a bunch of processes will be executed, and what to do if new processes arrive during the execution of another one. The list of initial processes, and the processes that will arrive later will be provided to you. This section is for the description of the concepts. The implementation details will be provided in the upcoming sections.

The processes in an operating system come in a hierarchy. More clearly, there are parent-children relationships between processes that leads to building of a process tree. In our system, these relationships are limited to be linear, and all the subprocesses related to a parent process must be stored in a process **queue**. Do not let the name fool you, this is called a queue only because it is a FIFO data structure. Clearly, these process queues are to hold linked processes. Note that, there will be different parent processes, i.e. all the processes does not have to be related, so likely there will be more than one process queues for an initial setup.

As you get the incoming processes in appropriate way, you will store them in the process manager, which is a **double ended queue (deque)**. The critical part about this deque is it must preserve the priority difference between the listed processes.

The failures are a natural and crucial part of the life, and of course, there will be failures in process management as well. During the execution phase, the failed processes must be stored somewhere, for them to be executed again in another time. For this assignment, this somewhere is a failure **stack** that you will implement.

Last but not least, since you will be provided with the information of the processes that will arrive during the execution stage, you will need another data structure to store them. This data structure is another **queue**, which is called insertion queue.

1.3. Working Scenario

In this section, you will get a more detailed scenario for the system run. Firstly, you will be given two different txt files: *initial_processes.txt* and *arriving_processes.txt*.

The *initial_processes.txt* includes the information of the processes that will be loaded to the process manager initially. There are three columns in it:

- **PID:** Id of the process, integer
- **Priority:** Priority of the process, integer 1 or 0 for high or low respectively.
- **IsHead:** Indicates whether the process is the parent of a process queue or not. If 1, the process is the parent. If 0, it is not. During the reading of the file, you need to enqueue the processes with 0 isHead value to a single process queue until you see a line with isHead value 1, which will be the last process to be enqueued to that queue. After this, that process hierarchy is completed and you can insert it to the process manager with respect to its priority. If this sounds complicated, do not worry, there will be a sample run to clarify in the upcoming sections.

You will implement a function to read this file and prepare the process manager (deque) for the execution loop.

The *arriving_processes.txt* includes the information of the processes that will arrive to the manager during the execution loop. There are four columns in it. The three are the same as the other txt file. The column special for this file is:

- **Iteration:** Indicates the number of iteration that the process is set to arrive to the process manager, integer.

The processes will be ordered by the *iteration* value. The processes related to each other will always arrive in the same iteration. You will implement a function to read this file as well to prepare an insertion queue, which will be used in the execution loop to insert new coming processes.

After reading both files and preparing the necessary data structures, you are ready to start an execution loop. In every iteration of this loop, the process on the high-priority end of the process manager will be executed. The related processes must be executed one after another, i.e. no other process can be executed between two related processes. For this assignment, there is a preset process failure case, which is the PID modulo 8 being 0. The failed processes must be pushed to failure stack. Note that if a process in a process fails, all of the related processes coming after it will not be executed as well. The arriving processes stored in the insertion queue will be inserted to the process manager in appropriate iterations. This loop will finish when there are no processes left in the process manager. To sum up, in each iteration, one process will be executed. If there is any arriving process for that iteration, that will be inserted to the process manager. And if the process fails, it will be pushed to failure stack.

Again, if you get confused, you can refer to sample run.

1.4. Data Structures and Functions

You will implement a process structure and four other data structures to whold processes for different use cases:

```c
typedef struct {
    int pid;
    int priority;
} PROCESS;

void initialize_process(PROCESS *p, int pid, int priority);
```

Listing 1.1: Process Structure

```c
typedef struct {
    PROCESS queue[QUEUE_SIZE];
    int front;
    int rear;
    int size;
    int priority;
    int iteration; // Necessary for process additions during execution
} PROCESS_QUEUE;

void initialize_process_queue(PROCESS_QUEUE *pq);

bool isFull(PROCESS_QUEUE *pq);

bool isEmpty(PROCESS_QUEUE *pq);

PROCESS peek(PROCESS_QUEUE *pq);

void enqueue(PROCESS_QUEUE *pq, PROCESS data);

PROCESS dequeue(PROCESS_QUEUE *pq);
```

Listing 1.2: Process Queue Structure

```c
typedef struct {
    PROCESS_QUEUE deque[MAX_PROCESS];
    int front;
    int rear;
    int size;
} PROCESS_MANAGER;

void initialize_process_manager(PROCESS_MANAGER *pm);

bool isFull(PROCESS_MANAGER *pm);

bool isEmpty(PROCESS_MANAGER *pm);

void insert_front(PROCESS_MANAGER *pm, PROCESS_QUEUE pq);

void insert_rear(PROCESS_MANAGER *pm, PROCESS_QUEUE pq);

PROCESS_QUEUE delete_front(PROCESS_MANAGER *pm);

PROCESS_QUEUE delete_rear(PROCESS_MANAGER *pm);
```

Listing 1.3: Process Manager Deque Structure

```c
typedef struct {
    PROCESS_QUEUE stack[MAX_FAILED];
    int top;
} FAILURE_STACK;

void initialize_failed_stack(FAILURE_STACK *fs);

bool isFull(FAILURE_STACK *fs);

bool isEmpty(FAILURE_STACK *fs);

void push(FAILURE_STACK *fs, PROCESS_QUEUE data);

PROCESS_QUEUE pop(FAILURE_STACK *fs);
```

Listing 1.4: Failure Stack Structure

```c
typedef struct {
    PROCESS_QUEUE queue[MAX_OPERATION];
    int front;
    int rear;
    int size;
} INSERTION_QUEUE;

void initialize_execution_queue(INSERTION_QUEUE *iq);

bool isFull(INSERTION_QUEUE *iq);

bool isEmpty(INSERTION_QUEUE *iq);

PROCESS_QUEUE peek(INSERTION_QUEUE *iq);

void enqueue(INSERTION_QUEUE *iq, PROCESS_QUEUE data);

PROCESS_QUEUE dequeue(INSERTION_QUEUE *iq);
```

Listing 1.5: Insertion Queue Structure

In addition to these structures, you will implement three functions for the working scenario of the process management system:

```c
// Function to read initial processes file
void read_process_file(const char *filename, PROCESS_MANAGER *pm);

// Function to read insertion file
void read_insertion_file(const char *filename, EXECUTION_QUEUE *eq);

// Function for the execution loop
void execution_loop(PROCESS_MANAGER *pm, EXECUTION_QUEUE *eq,
    FAILED_STACK *fs);
// For this function, you need to provide system output in the format
    stated in the sample run.
```

Listing 1.6: Process Manager System Functions

2. Sample Run

Let’s simulate a scenario by walking through a sample input to clarify the assignment better. As explained above, there will be two text files for the input. In this section, you will see a proper input-output match.

Sample Input Initial Processes File

File Name: `initial_processes.txt`

```
pid, priority, isHead
120, 1, 1
131, 1, 0
132, 1, 0
130, 1, 1
140, 0, 1
151, 1, 0
150, 1, 1
168, 0, 0
160, 0, 1
```

Sample Input Insertion Info File

File Name: `arriving_processes.txt`

```
iteration, pid, priority, isHead
1, 200, 0, 1
3, 211, 1, 0
3, 212, 1, 0
3, 210, 1, 1
8, 220, 0, 1
```

Sample Output File - s:success f:fail

File Name: `execution_run.txt`

```
151, s
150, s
131, s
132, s
130, s
211, s
212, s
210, s
120, f
140, s
168, f
200, f
220, s
```

In this scenario, first the process manager is loaded with the processes read from the *initial_processes.txt* file. The values with high priority are inserted the one end of the deque, and the rest are inserted to the other end. After the file is read, the process manager deque will look like this:

```
<- 151 - 131 - 120 - 140 - 168 ->
    |     |                 |
   150   132               160
          |
         130
```

The processes shown as columns represent the related processes. Here, the leftmost end is the high priority, and the rightmost end is the low priority. So, the execution will start from the leftmost part. The arrows are put there to show that this is a deque.

Then, the arriving processes are read from the *arriving_processes.txt*. The insertion queue data structure is a regular queue. So the processes will be enqueued as they come in the text file. After the file is read, the insertion queue will look like this:

```
<- 200 - 211 - 220
          |
         212
          |
         210
```

The iteration numbers start from zero. In this sample run, since the related processes are executed successively, they are shown as *to be executed* after one of them starts executing, to prevent confusion. After each iteration, the process manager will look like this:

```
Iter 0: 151 executed
Process Manager:
<- 131 - 120 - 140 - 168 ->
    |                 |
   132               160
    |
   130
To be executed: 150
Insertion Queue:
<- 200 - 211 - 220
          |
         212
          |
         210
Failure Stack: -
---------------------------------------
Iter 1: 150 executed, 200 arrived
Process Manager:
<- 131 - 120 - 140 - 168 - 200 ->
    |                 |
   132               160
    |
   130
To be executed: -
Insertion Queue:
<- 211 - 220
    |
   212
    |
   210
Failure Stack: -
---------------------------------------
Iter 2: 131 executed
Process Manager:
<- 120 - 140 - 168 - 200 ->
                |
               160
To be executed: 132 - 130
Insertion Queue:
<- 211 - 220
    |
   212
    |
   210
```

```
Failure Stack: -
---------------------------------------
Iter 3: 132 executed, 211 - 212 - 210 arrived
Process Manager:
<- 211 - 120 - 140 - 168 - 200 ->
    |                 |
   212               160
    |
   210
To be executed: 130
Insertion Queue:
<- 220
Failure Stack: -
---------------------------------------
Iter 4: 130 executed
Process Manager:
<- 211 - 120 - 140 - 168 - 200 ->
    |                 |
   212               160
    |
   210
To be executed: -
Insertion Queue:
<- 220
Failure Stack: -
---------------------------------------

Nothing exciting, regular iterations

---------------------------------------
Iter 8: 120 executed, failed, 220 arrived
Process Manager:
<- 140 - 168 - 200 - 220 ->
          |
         160
To be executed: -
Insertion Queue: -
Failure Stack: <- 120
---------------------------------------
Iter 9: 140 executed
Process Manager:
<- 168 - 200 - 220 ->
    |
   160
To be executed: -
Insertion Queue: -
Failure Stack: <- 120
---------------------------------------
```

```
Iter 10: 168 executed, failed
Process Manager:
<- 200 - 220 ->
To be executed: -
Insertion Queue: -
Failure Stack: <- 168 - 120
                   |
                  160
---------------------------------------
Iter 11: 200 executed, failed
Process Manager:
<- 220 ->
To be executed: -
Insertion Queue: -
Failure Stack: <- 200 - 168 - 120
                         |
                        160
---------------------------------------
Iter 12: 220 executed
Process Manager: -
To be executed: -
Insertion Queue: -
Failure Stack: <- 200 - 168 - 120
                         |
                        160
--------------------------------------- DONE!
```

3. Deliverables for Successful Completion

You will be provided with the header files necessary for the implementations of the data structures, and the necessary test files. Your task is to complete the implementation by writing the source files for each header file. You can call the process manager system functions in the main.cpp file. You will also be provided with sample test files.

Feel free to modify main.cpp to view intermediate results as you work through the problem. However, keep in mind that while you may adjust main.cpp, I will be testing your solution with a different version of the file.

Also note that the provided test files will not be the only ones used for grading. Your code will be evaluated on additional test cases, so make sure your implementation is robust and can handle various scenarios. You can try and write your own test cases for this purpose. Also, you are strongly motivated to test your code with different input files that you may create.

Good luck! For any questions, feel free to e-mail me.

IMPORTANT NOTE:

- Copying code fragments from any source, including books, websites, or classmates, is considered plagiarism.
- You are free to conduct research on the topics of the assignment to gain a better understanding of the concepts at an algorithmic level.
- Refrain from posting your code or report on any public platform (e.g., GitHub) before the deadline of the assignment.
- You are **NOT** permitted to use the STL for any purposes in this assignment.
- Remember this course is taught in C language, C++ usage is strictly restricted to I/O operations.
- You **must** stick with the container environment that is provided to you. This is crucial for the grading of your work. Any code that is **not** compiled inside the container will receive **zero** grade.
- Additionally, please refrain from using any AI tools to help you complete your work. While I cannot directly detect AI-generated code, relying on such tools is not in your best interest. Everything you learn in this class forms the foundation for future courses, and if you do not build a strong foundation now, you will likely struggle in the semesters to come.