Warm-Up Assignment: Kernel Data Structures

This assignment is designed to provide students with a practical and in-depth understanding of fundamental data structures that are integral to the design and implementation of modern operating system kernels. Specifically, you will implement and work with the following data structures in C: **Stack**, **Circular Queue**, **Circular Linked List**, and **Min-Priority Queue**. These data structures are not only foundational from a theoretical computer science perspective, but they are also critically important in low-level systems programming and operating system kernel development, where deterministic behavior, minimal overhead, and optimal use of constrained resources are paramount.

TA (grader) Contact Details:

For any questions or concerns related to assignment grading, please reach out to the TA listed below. Ensure that all communication is polite and respectful. You must contact the TA first for any grading issues. If the matter remains unresolved, feel free to reach out to me.

- **TA Name:** Behzad Ousat
- **Email:** bousa001@fiu.edu

1. Relevance of aforementioned data structures in OS Kernel Design

- **Stack**: Used extensively in kernel space for function call management, interrupt handling, and managing process contexts. Kernel-mode stacks are typically small and require efficient, predictable behavior, making it important to understand stack operations at a low level.
- **Circular Queue**: Commonly used in producer-consumer problems such as scheduling queues, interrupt buffers, and I/O buffer management. It provides a fixed-size, efficient queue structure ideal for ring-buffer-based communication within kernel subsystems.

- **Circular Linked List**: Employed in the management of process scheduling (e.g., round-robin schedulers), maintaining device queues, or managing active resources in a cyclic fashion. Their pointer-based structure allows for dynamic memory usage while still supporting circular traversal patterns required in many OS services.
- **Min-Priority Queue**: Integral to process scheduling (e.g., priority scheduling), disk I/O scheduling, and timer management. It provides efficient access to the element with the highest urgency (lowest key), which is a fundamental requirement in real-time and priority-aware systems.

2. ⚠️ IMPORTANT: Your application output must match exactly with the output of the instructor-provided sample executable (i.e., the expected output)!

- An **automated grading system (autograder)** will be used to compare your program's output against the expected output.

  > **Note:** Even minor differences—such as an extra space, a missing newline, or a single incorrect character—will result in your output being marked **incorrect**.

- To generate the `expected_output.txt` file from the instructor-provided sample executable `A1_sample`, use the following command. The `>` symbol is the redirection operator that sends the output of the executable to the specified file:

  ```
  ./A1_sample > expected_output.txt
  ```

- To ensure accuracy, you are strongly encouraged to use a **file comparison tool** to compare **your implementation's output** with the **expected output** before submission. One recommended option is:

  🔗 **Mergely** – Compare files and find differences online

  ➡️ *Click the **'diff files online'** button in the top-right corner to enter full comparator mode, if not taken there automatically.*

- Use this tool to verify that your output matches the expected output **byte-for-byte**.

3. Project Framework and Submission Instructions

This assignment provides a structured framework to guide you through implementing **four** essential data structures used in operating system kernels. The framework enforces modularity and real-world OS design patterns, mimicking how kernel code is organized and built.

a) Provided Files

You are given the following files as part of the starter code. Please ensure that **all the files listed below are placed in the same directory as** `driver.c` in order to successfully **compile and test** your executable.

- `driver.c`: A prewritten test driver containing test logic for each data structure.
  🔒 **Do NOT modify this file.**
- `a1_header.h`: The header file containing shared data structure definitions and function prototypes.
  🔒 **Do NOT modify this file.**
- **Five skeleton `.c` modules** for your implementation:
  - `stack.c`
  - `circular_queue.c`
  - `circular_linked_list.c`
  - `min_priority_queue.c`
- `Test Cases`: The following three files contain **simple**, **moderate**, and **rigorous** test cases designed to validate the operations of all data structures.
  - *simple_testcase.txt*
  - *moderate_testcase.txt*
  - *rigorous_testcase.txt*

  💀The `driver.c` program reads test cases from a file named `TESTCASES.txt`. During your testing, you are required to **copy and paste the relevant test cases** from `simple_testcase.txt`, `moderate_testcase.txt`, and `rigorous_testcase.txt` into `TESTCASES.txt` based on the data structure you are currently implementing.

⚠️ **Important**: These are example test cases; final evaluation will involve **more complex scenarios**. If you can pass all these three testcases, you can easily pass the instructor grading test cases!

- `A1_sample`: This is the instructor-provided executable that demonstrates the **correct implementation** of all data structures and their operations. Use it in conjunction with the provided **test cases** to thoroughly understand the expected behavior. Your goal is to **replicate the functionality exactly** as demonstrated by this executable.

  ⚠️ **Important**: Your program's output must **exactly match** the output produced by the instructor-provided executable (`A1_sample`) for equivalent input. Any deviation in **behavior**, **logic**, or **output formatting** may result in **point deductions** during grading. You are strongly advised to test your implementation thoroughly and validate it against a wide range of inputs—not just the provided sample test cases.

✅ *Execution Permissions Notice:*

If you do **not** have execute permissions for the instructor-provided `A1_sample` executable, you can manually grant the required permissions using the following command on a **Linux system**.

> **Note:** When files are uploaded or transferred to the **Ocelot server**, execute permissions from the users **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

🛠️ **Granting Permissions**

```
# Check current file permissions
ls -l <filename>

# If you see:
# -rw-------  OR encounter "Permission denied" on execution

# Run the following command to grant permissions:
chmod +rwx <filename>
```

Replace `<filename>` with the actual name of the executable (e.g., `A1_sample`) to ensure it is readable, writable (if necessary), and executable on the server.

Provided Files Layout:

```
A1_PROVIDED_FILES/
├── Framework/
  - stack.c  #Skeleton for Stack (implement it)
  - circular_queue.c  #Skeleton for Circular Queue (implement it)
  - circular_linked_list.c  #Skeleton for Circular Singly Linked List
(implement it)
  - min_priority_queue.c  #Skeleton for Min-Heap based Priority Queue
(implement it)
  - driver.c  #Instructor-provided driver (do not modify!)
  - a1_header.h  #Instructor-provided header (do not modify!)
  - TESTCASES.txt  # Driver reads the testcases from this file (Copy-paste
the testcases into it)
├── Testcases/
  - simple_testcase.txt  #Basic validation tests (copy-paste into
TESTCASES.txt for tesitng)
  - moderate_testcase.txt  #Intermediate-level tests (copy-paste into
TESTCASES.txt for tesitng)
  - rigorous_testcase.txt  #Comprehensive stress tests(copy-paste into
TESTCASES.txt for tesitng)
├── Sample_Executable
  - A1_sample  #Reference executable from instructor. Experiment with it!
├── README.txt  #You must submit the team details in the same format
```

b). Compilation Requirement – Makefile

You must write and submit a `Makefile` that compiles all your `.c modules` along with `driver.c` and `a1_header.h` into a single executable named **`datastructure`**.

Makefile Rules:

- Use `gcc` with `-Wall -Wextra -pedantic` to ensure professional-grade code quality and compliance with strict C standards.
- Build object files and link them together.
- Provide a `clean` rule to remove object files and the final binary.

🤔 How to Implement and Test Your Application Professionally?

1. Plan Before You Code

- Understand the problem requirements *thoroughly*, and review the `Assignment Specification`, `Test Cases`, and the instructor-provided `Sample Executable` *carefully.*
- Review and thoroughly understand the required data structures and their associated operations before beginning your implementation.

2. Implement Incrementally

- Start by implementing *one data structure at a time*.
- Ensure that each structure and its operations are *fully functional before moving to the next*.

3. 💡 Isolate and Test Early

- Implement **one data structure at a time**.
- Test your implementation **progressively**, starting with:
    - **Simple** test cases
    - Then **moderate** test cases
    - And finally, **rigorous** test cases
- Once your implementation successfully passes **all three levels of test cases**, proceed to the **next data structure** and repeat the same testing process.

This approach ensures thorough validation and minimizes debugging complexity as you build your solution.

4. Progressively Increase Test Complexity

- Once the simple case passes, move on to the *moderate*, and then the *rigorous* test case.
- These are example test cases; final evaluation will involve **more complex scenarios**.

5. Validate Against the Sample Executable

- Use the instructor-provided `A1_sample` to run test cases and understand the expected output.

- Ensure your output matches **exactly**, including spacing, formatting, and content.

6. Maintain Clean, Modular Code

- Keep your code organized with proper function definitions and comments.
- ⚠ **Avoid hardcoding** values or paths.

7. Perform Final Verification

- Re-run all test cases after completing your implementation.
- Confirm that your application behaves consistently and correctly in all scenarios.

✅ YOU MUST

Test your final implementation on the **Ocelot server** as well as other systems to ensure portability and to identify any potential issues, such as dependency conflicts, architecture differences, or compiler compatibility problems.

4. Data Structures and Their Operations to Be Implemented

a). Stack (LIFO)

Definition: A linear data structure where elements are added and removed from the same end, known as the "top".

Operations to Implement:

- `init_stack(stack)`: Initializes the stack by setting the `top` index to `-1`, indicating that the stack is currently empty.
- `push(stack, element)`: Adds `element` to the top of the stack, otherwise returns `-1` if stack is full.
- `pop(stack)`: Removes and returns the `top` element from the stack, otherwise returns `-1` if stack is empty.
- `peek_stack(stack)`: Return top element if stack is not empty, `-1` otherwise.
- `is_stack_empty(stack)`: Returns `1` if the stack has no elements, otherwise returns `0`.

- `is_stack_full(stack)`: Returns `1` if the stack has reached its maximum capacity, otherwise returns `0`.
- `stack_size(stack)`: Returns number of elements currently in the stack.
- `list_stack_elements(stack)`: Prints the elements currently in the stack, otherwise returns `-1` if stack is empty.

---

b). Circular Queue (FIFO)

Definition: A linear data structure where elements are inserted from the rear and removed from the front.

Operations to Implement:

- `init_queue(queue)`: Initializes the circular queue by setting both `front` and `rear` pointers to `-1`, marking it as empty.
- `enqueue(queue, element)`: Inserts `element` at the rear of the circular queue and returns `0` on success, otherwise `-1` if queue is full.
- `dequeue(queue)`: Removes and returns the element at the `front` of the circular queue, otherwise returns `-1` if queue is empty.
- `peek_queue(queue)`: Returns the element at the `front` of the queue without removing it. Returns `-1` if the queue is empty.
- `is_queue_empty(queue)`: Returns `1` if empty, `0` otherwise
- `is_queue_full(queue)`: Returns `1` if full, `0` otherwise.
- `queue_size(queue)`: Returns number of elements in queue.
- `list_queue_elements(queue)`: Prints the elements currently in the queue, otherwise returns `-1` if queue is empty.

c). Circular Linked List

Definition: A linked list in which the last node points back to the first node, forming a circle.

Operations to Implement:

- `create_node(data)`: Allocates and returns a new node containing the specified `data`, otherwise exit application.

- `is_list_empty(head)`: Returns `1` if the list is empty, `0` otherwise.
- `insert_node(head, new_node)`: Inserts `new_node` into the circular linked list.
- `delete_node(head, key)`: Searches for and deletes the `node` with the specified `key`. Returns the potentially updated `head` pointer. Returns NULL if the list was empty or had only one node (which is now deleted).
- `search_node(head, key)`: Searches for a `node` containing `key` in the list. Returns a pointer to the matching `node` if found, otherwise returns `NULL`.
- `iterate_list(head)`: Traverses the entire circular linked list starting from `head` and prints the data of each `node` exactly once. Returns `-1` if list is empty, `0` otherwise.
- `free_list(head)`: Traverses the entire circular linked list starting from `head` and deletes each `node` along the way. Returns `-1` if list is already empty, `0` if memory was successfully freed.

d). Min-Priority Queue / Min-Heap

Definition:
 A data structure in which each element is associated with a numeric priority, and elements are served **based on the lowest priority value** — i.e., **lower numbers indicate higher priority**.

Operations to implement:

- `insert_element(heap, element)`: Inserts a new `element` into the min-heap with an associated priority. Simply `returns` if the Heap is full.
- `extract_min(heap)`: Removes and returns the `element` with the **lowest** priority value from the `heap` (i.e., the highest priority task). The `heap` is restructured afterward.
- `peek_heap(heap)`: Returns the `element` with the **lowest** priority without removing it.
- `decrease_key(heap, element_index, new_priority)`: Lowers the priority of the `element` at `element_index` to `new_priority`. If the new priority is lower than the parent's, the `heap` property is restored by promoting the element upward.
- `remove_element(heap, element_index)`: Deletes the `element` at `element_index` from the `heap`. Internally uses `decrease_key` followed by `extract_min` to ensure proper restructuring.

- `min_heapify(heap, index)`: Maintains the min-heap property starting from the node at `index`, moving downward through the tree to restore order after removal or key update.
- `is_heap_empty(heap)`: Checks whether heap is empty or not. Returns `1` if the heap is empty, `0` otherwise.
- `list_heap_elements(heap)`: Prints the elements currently in the heap. Returns `-1` if error, `0`if heap is empty.

---

5. Deliverables Folder Structure

You are required to **strictly follow** the folder and file structure outlined below when submitting your assignment. Any deviation from this format **WILL** result in penaltiest.

⚠️ **Important**: Your submission must be a zip file named:

`XXX.zip`

(Refer to the next section for the submission folder naming convention to replace `xxx` appropriately)

Submission Folder Layout:

```
XXX.zip
├── Submissions/
  — *.c  #All your module (.c) source files
  — a1_header.h #Intructor provided header file (must not be modified)
  — driver.c  #Instructor provided driver file (must not be modified)
  — makefile  #Must support 'make' and'make clean'(ZERO credit if non-
compliant or missing)
  — datastructures #Executable generated by your Makefile (for backup only,
not used for grading)
├── README.txt  #Team details (see the below section)
```

6. Assignment Submission guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!

⚠️ The filename must follow the below format **considering the team's corresponding member name**:
`A1_<FirstName_LastName>.zip` For example: **A1_James_Bond.zip**

- Due dates will be posted in the **Assignments** section on Canvas.
- ⚠️ **Important:** For **group submissions**, each group must designate one `corresponding member`. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
- Submission **must include a** `Makefile` that successfully compiles your implementation to produce an executable. **Submissions without a Makefile will receive a grade of zero.**
- ⚠️ **Important:** Your implementation **must be compiled and tested on the Ocelot server** to ensure correctness and compatibility. Submissions that fail to compile or run as expected on Ocelot will receive a grade of `ZERO`.
- Whether submitting as a **group or individually**, every submission must include a `README` file containing the following information:
  1. Full name(s) of all group member(s).
  2. **Prefix the name of the corresponding member with an asterisk** `*`.
  3. Panther ID (PID) of each member.
  4. FIU email address of each member.
  5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**
- ⚠️ **Important - Late Submission Policy:**
  For each day the submission is delayed beyond the `Due date` and until the `Until date` (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until Date will not be accepted or considered for grading.**
- Please refer to the course syllabus for any other details.

7. Grading Rubric

Criteria	Marks
⚠️ **No Makefile**	**0**

Criteria	Marks
⚠️ **If the Makefile fails to generate the executable, or if your executable cannot be run or tested**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 70**

Detailed Breakdown of 100 Marks

Category	Marks
1. Stack Implementation	20
2. Circular Queue Implementation	20
3. Circular Linked List Implementation	20
4. Min-priority Queue (Min-heap	20
5.Clear code structure, Meaningful comments, well-written README, Overall readability	20
Total of (5)	**100**

++++++++++++++++++++++ ALL THE BEST ++++++++++++++++++++++