📚 COP4610 Assignment 1: Kernel Data Structures Implementation

📄 Table of Contents

📢 Important Note!

Please ensure that your implementation works on the course-designated ocelot server: ocelot-bbhatkal.aul.fiu.edu . Claims such as **"it works on my machine"** will not be considered valid, as all testing and grading will be conducted in that environment.

⚠️ **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes all three test cases— simple, moderate, and **rigorous**—using the autograder, you are likely to receive full credit for your submission.

1. 📧 TA Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

Communication Guidelines:

- 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

Name: Angie Martinez — amart1070@fiu.edu
Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah — svisw003@fiu.edu
Sections: COP 4610-U01 (84957), COP 4610-UHA (85236)

2. 📖 Assignment Overview

This assignment provides **hands-on experience** with fundamental **data structures**, **kernel-level programming concepts**, and **core systems programming principles**. You will implement **five essential data structures** that form the backbone of **modern operating system kernels**—the same types of structures running right now in the kernel of your laptop, smartphone, and the servers powering the internet.

Through this implementation, you will explore how operating systems **schedule processes**, **allocate memory**, **queue I/O requests**, and **manage system resources**. Your implementation must exhibit **deterministic behavior** — for the same input, it must consistently produce the same output, **minimal overhead**, and **optimal performance** in constrained environments—just as kernel-level code must in real-world systems.

These kernel data structures are not abstract academic concepts—they are the **critical infrastructure** enabling every application you run, every file you save, and every network request you make. Mastering their implementation will deepen your understanding of how operating systems achieve reliability, efficiency, and scalability at the lowest levels of the computing stack.

3. 🛠️ Skills You'll Develop

In this assignment, you'll **gain hands-on experience** with both the **technical skills** and **operating system concepts** that are essential for professional systems programming and kernel-level development.

- **Implement** core kernel data structures efficiently (stack, queue, linked list, heap, bitmap).
- **Manage** memory dynamically with proper allocation, cleanup, and leak prevention.
- **Work** with complex pointers, including circular references and heap navigation.
- **Optimize** algorithms for speed and memory efficiency in resource-limited environments.
- **Write** deterministic, predictable code for system-critical functionality.
- **Use** modular design principles for maintainable and reusable code.
- **Test** your code thoroughly against edge cases and performance benchmarks.
- **Produce** professional-quality C code following industry standards.

4. 🎯 Learning Objectives

- **Understand** and implement data structures used inside real OS kernels.
- **Apply** scheduling algorithms using priority queues and circular lists.
- **Use** stacks for process management and function call tracking.
- **Design** circular buffers for efficient I/O and producer-consumer workflows.

- **Track** and manage resources dynamically, just like an OS does.
- **Optimize** insert, delete, and lookup operations for high performance.
- **Build** robust error handling to maintain stability in critical systems.

---

5. ⚙️ Assignment Tasks

Data Structure Implementation Requirements

You will implement **five fundamental data structures** using a provided modular framework:

1). Stack (LIFO) - Last In, First Out:

A linear data structure where elements are added and removed from the same end (the "top")

- **Core Operations**: `init_stack`, `push`, `pop`, `peek_stack`, `is_stack_empty`, `is_stack_full`, `stack_size`, `list_stack_elements`
- **Usage in OS**: Function call management, interrupt handling, process context switching
- **Performance**: O(1) for all operations

2). Circular Queue (FIFO) - First In, First Out:

A linear data structure with circular wrap-around where elements are inserted at rear and removed from front

- **Core Operations**: `init_queue`, `enqueue`, `dequeue`, `peek_queue`, `is_queue_empty`, `is_queue_full`, `queue_size`, `list_queue_elements`
- **Usage in OS**: Process scheduling queues, I/O buffering, producer-consumer scenarios
- **Performance**: O(1) for all operations with efficient space utilization

3). Circular Linked List:

A dynamic data structure where the last node points back to the first, forming a circular chain

- **Core Operations**: `create_node`, `is_list_empty`, `insert_node`, `delete_node`, `search_node`, `iterate_list`, `free_list`
- **Usage in OS**: Round-robin scheduling, device management, dynamic resource tracking
- **Performance**: O(1) for insertion/deletion, O(n) for search operations

4). Min-Heap Priority Queue:

A complete binary tree where each parent has lower priority value than its children (lower numbers = higher priority)

- **Core Operations**: `insert_element`, `extract_min`, `peek_heap`, `decrease_key`, `remove_element`, `min_heapify`, `is_heap_empty`, `list_heap_elements`
- **Usage in OS**: Process scheduling, timer management, disk I/O scheduling
- **Performance**: O(log n) for insert/extract, O(1) for peek operations

5). Bitmap:

A space-efficient data structure using individual bits to represent boolean states

- **Core Operations**: `init_bitmap`, `set_bit`, `clear_bit`, `test_bit`, `find_first_zero_bit`, `find_next_set_bit`, `print_bitmap`, `is_bitmap_empty`, `bitmap_size`
- **Usage in OS**: Page frame tracking, CPU masks, process scheduling, resource allocation
- **Performance**: O(1) for set/clear/test operations, O(n) for find operations

Implementation Standards

- Each data structure must handle **identical test input** and produce **deterministic output**
- Implement **robust error handling** for edge cases (empty structures, full structures, invalid operations)
- Use **MAX_SIZE = 100** for fixed-size structures (Stack, Circular Queue, Min-Heap)
- Ensure **memory safety** with proper allocation/deallocation for dynamic structures
- Follow **exact function signatures** provided in skeleton files for autograder compatibility

---

6. 🏗 Project Framework

This assignment includes a comprehensive development framework comprising **skeleton code files** (*templates for your implementation*), **utility functions** (*if applicable*), a **driver program**, a **Makefile**, and an **autograder** to support and guide your implementation process.

1) Provided Framework Infrastructure (*Do Not Modify*)

- `ds_header.h`: Complete header file with data structures, constants, and function prototypes.**(study it thoroughly!)** .
- `driver.c`: The main driver program responsible for coordinating all tests and managing the overall execution flow.
- `Makefile`: Used to build your application and generate the final executable `kernelDs`. You can run it using the following command:

```
./kernelDS
```

- `autograder_kernelDS.sh`: Automated testing script **(identical to the final grading mechanism)**.

2) Student Implementation Files

- `skeleton_stack.c` : Stack implementation template with guided TODO sections. Rename it to → `stack.c`.
- `skeleton_circular_queue.c` : Circular queue template with guided TODO sections. Rename it to → `circular_queue.c`
- `skeleton_circular_linked_list.c` : Linked list template with guided TODO sections. Rename it to → `circular_linked_list.c`
- `skeleton_min_heap.c` : Min-heap template with guided TODO sections. Rename it to → `min_heap.c`
- `skeleton_bitmap.c` : Bitmap template with guided TODO sections. Rename it to → `bitmap.c`

3) Testing Infrastructure

- `TESTCASES.txt`: # Test cases (**copy simple/moderate/rigorous testcases here**).
- `EXPECTED_OUTPUT.txt`: # Expected results (generated by executing **./A1_sample > EXPECTED_OUTPUT.txt**).
- **Test Case Sets:** Multiple test case sets ranging from simple to rigorous complexity. As you progress with your implementation, copy each of these test cases into `TESTCASES.txt` one by one. **You are required to pass all the test cases** !
    - `testcases_simple.txt` / `expected_output_simple.txt`: Basic validation tests.
    - `testcase_moderate.txt` / `expected_output_moderate.txt`: Intermediate complexity tests.
    - `testcase_rigorous.txt` / `expected_output_rigorous.txt`: Comprehensive stress tests.

4) ⭐ Implementation Guidance Documentation

- `A1_pseudocode_and_examples.pdf`: Comprehensive implementation guide with detailed pseudocode, step-by-step examples, and algorithm explanations for all data structures
    - Study this document before implementing each data structure
    - Contains detailed examples demonstrating expected behavior
    - Provides time/space complexity analysis and optimization guidance
    - Cross-referenced with test cases for validation

👍 **Help**: **Use the pseudocode guide in conjunction with skeleton files for optimal implementation success.**

5) Instructor's Reference Implementation

`A1_sample`: The instructor-provided reference executable that demonstrates the **complete, correct, and required implementation** of this assignment. 💡 **This executable serves as your primary reference for understanding the expected behavior.**

⚠️ **Important**: Your program's output must **exactly match** that of the instructor-provided executable for equivalent input. Any deviation in behavior, logic, or output formatting may prevent the autograder from evaluating your submission.

6) Data Structure Requirements

All data structures must use the definitions provided in `ds_header.h`. Refer to this header file **thoroughly** to understand the constants, global entities, function prototypes, and other framework components required for your implementation.

7) Provided Files Layout

```
A1_PROVIDED_FILES/
├── Sample_Executable/
    └── A1_sample                # Reference implementation (study this thoroughly!)
├── Framework/
    ├── ds_header.h              # Header file (provided - DO NOT MODIFY)
    ├── driver.c                 # Application driver (provided - DO NOT MODIFY)
    ├── Makefile                 # Builds your application (provided - DO NOT MODIFY)
    ├── autograder_kernelDS.sh   # Autograder (provided - DO NOT MODIFY)
    ├── TESTCASES.txt            # Test cases (copy simple/moderate/rigorous testcases here)
    ├── EXPECTED_OUTPUT.txt      # Expected results (get by ./A2_sample > EXPECTED_OUTPUT.txt)
    └── *skeleton_files.c         # Your implementation files (rename to functions.c for submission)
├── Testcases/
    ├── testcases_simple.txt       # Simple testcases
    ├── testcase_moderate.txt      # Moderate testcases
    └── testcase_rigorous.txt      # Rigorous testcase
├── A1_Specification.pdf           # This detailed specification & requirement document
├── A1_Testing_Guide               # A comprehensive tesing guide
├── A1_pseudocode_and_examples.pdf # Implementation guide with detailed pseudocod
└── README.txt                     # You must submit team details in the same format
```

8) ✅ Execution Permissions Notice

If you **do not** have execute permissions for the instructor-provided `A1_sample`, `ds_header.h`, `autograder_kernelDS.sh` and any other files, you can manually grant the required permissions using the following command on a **Linux system**.

> **Note:** When files are uploaded or transferred to Ocelot server (`ocelot-bbhatkal.aul.fiu.edu`), execute permissions **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

Granting Permissions to All Items in the Current Directory

```
# Check current permissions
ls -l

# Grant read, write, and execute permissions (safe and preferred - applies only what's missing)
chmod +rwx *

# Assign full permissions (read, write, execute) to all users for all files in the current directory
# ⚠️ Not preferred - use only as a last resort!
chmod 777 *

# Verify that permissions were applied
ls -l
```

⚠️ **Caution: Use `chmod 777` with care!**
Granting full permissions to everyone can:

- Expose sensitive data
- Allow unintended modifications or deletions
- Introduce security risks, especially in shared or multi-user environments

8) ☑️ YOU MUST!

- **Test your implementation thoroughly** on the **Ocelot** server (`ocelot-bbhatkal.aul.fiu.edu`)
- **Ensure exact output matching** with the instructor sample executable `A1_sample`
- **Validate all calculations** and logic against the sample
- **Test edge cases** and error handling extensively
- **Verify compilation** with the specified compiler flags

7. 🔧 Development & Testing

Required Files for Development

All the following files must be in the same directory:

1. **Framework files**: `driver.c`, `Makefile`, `ds_header.h`
2. **Your implementation files**: `stack.c`, `circular_queue.c`, `circular_linked_list.c`, `min_heap.c`, `bitmap.c`
3. **Testing tools**: `autograder_kernelDS.sh`, `TESTCASES.txt`, `EXPECTED_OUTPUT.txt`

⚠ **Critical Testing Information:** **The instructor will use different and more rigorous test cases for final grading that are not provided to students.** **However, if you pass all provided rigorous test cases with 100% accuracy, you are highly likely to pass the instructor's final grading test cases..**

Comprehensive Development Workflow

> ⚠ **Important**: **Please refer to the document A1_Testing_Guide.pdf for the detailed testing instructions.**

Step 1: Understand Expected Output Format

```
# Generate expected output from instructor sample
./A1_sample > EXPECTED_OUTPUT.txt

# View the expected output format to understand correct behavior
cat EXPECTED_OUTPUT.txt
```

Step 2: Implement with Incremental Testing

- **Recommended Implementation Order:**
  - Stack → Circular Queue → Circular Linked List → Min-Heap → Bitmap
- **Test with Progressive Complexity:**
  - ☑ Please refer to the document A2_Testing_Guide.pdf for the detailed progressive tesing instructions.

Step 3: Use Autograder for Final Validation

- ⚠ **Important**: **Please note that instructor will use the same autograder for the final grading.**

Autograder System

The autograder compares your program's output with pre-provided reference output to determine correctness.

Internal Process:

1. **Clean rebuild**: `make clean && make`
2. **Runs your implementation**: `./kernelDS > STUDENT_OUTPUT.txt`
3. **Compares outputs**: Extracts each data structure section and compares with expected results

4. **Calculates scores**: Awards points based on percentage of matching output lines
5. **Provides detailed feedback**: Shows exactly which lines don't match

8. 📤 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 format: `A1_FirstName_LastName.zip` (e.g., **A1_Harry_Potter.zip**)
- ⚠️ **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
- ⚠️ **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.
- ⚠️ **Important Policy on Teamwork and Grading**

  All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene** .
- ⚠️ **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatkal.aul.fiu.edu`) **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.**
  - **Due Date**: ANNOUNCED ON CANVAS
  - **Until Date**: ANNOUNCED ON CANVAS
- Please refer to the course syllabus for any other details.

9. 📦 Deliverable Folder Structure

Upload to Canvas: Submit a single ZIP file named `A1_FirstName_LastName.zip`.

- You are **required to strictly follow** the folder and file structure outlined below when submitting your assignment.

- **Failure to adhere to this structure will cause the Instructor's batch autograder to fail in recognizing and extracting your submission, resulting in a grade of ZERO.**
- **You will be held responsible for receiving a ZERO grade if this guideline is not followed.**
- ⚠️ **For team submissions** – Include the first name and last name of the team's submitting member in the ZIP file name (e.g., `A1_Harry_Potter.zip`). **For solo submissions** – Use your own first and last name.

```
A1_FirstName_LastName.zip
├── stack.c                    # Your Stack implementation
├── circular_queue.c           # Your Circular Queue implementation
├── circular_linked_list.c     # Your Circular Linked List implementation
├── min_heap.c                 # Your Min-Heap implementation
├── bitmap.c                   # Your Bitmap implementation
└── README.txt                 # Team details (see submission guidelines)
```

10. 💯 Grading Rubric

Criteria	Points
⚠️ **Program fails to run or crashes during testing**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 70**

Detailed Breakdown of 100 Points

Grading Component	Points
1. `Stack` implementation	10
2. `Circular Queue` implementation	15
3. `Circular Linked List` implementation	15
4. `Min-Heap` implementation	25
5. `Bitmap` implementation	25
6. Clear code structure, Meaningful comments, well-written README, Overall readability	10
Total	**100**

🚀 ***Success Strategy:***
The student autograder uses the **same grading mechanism** as the final evaluation, awarding points based on the percentage of correct output matching for each data structure component.
The final **10 points** for code quality will be based on:

- Proper comments
- Clean structure
- Adherence to guidelines

- Overall readability

Tips for success:

- Study the reference implementation thoroughly
- Implement features incrementally with continuous testing
- Ensure **exact output matching**
- Maintain professional code quality throughout development

Good luck! 💪
