📚 COP4610 BONUS Assignment: Contiguous Memory Allocation (First-Fit, Best-Fit, Worst-Fit) with Memory Coalescing

🔑 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 the test case using the autograder, you are likely to receive full credit for your submission.

You must use these strict gcc compilation flags while building your application:
-std=c17
-D_POSIX_C_SOURCE=200809L
-Wall
-Wextra
-Werror
-g
-O0

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation with the provided testing framework on the Ocelot server before the final 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 Graders first for any grading issues**
- If the matter remains unresolved, feel free to reach out to the instructor

Name: Instructor – bbhatkal@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U01 (84957)

Name: Angie Martinez – amart1070@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah – svisw003@fiu.edu(⚠️ via the Canvas inbox only)
Sections: COP 4610-UHA (85236)

---

2. 📖 Assignment Overview

This assignment provides **hands-on experience** with **contiguous memory allocation**, **memory management strategies**, and **fragmentation control**. You will implement three fundamental **memory allocation algorithms** — **First-Fit**, **Best-Fit**, and **Worst-Fit** — along with **memory coalescing** to reduce fragmentation.

Through this implementation, you will explore how operating systems **allocate memory** to processes, **manage free memory blocks**, and **optimize memory utilization**. Your implementation must exhibit **deterministic behavior** – for the same input, it must consistently produce the same output, **correct memory allocation decisions**, and **proper handling of memory coalescing**—just as real operating system memory managers must.

These memory allocation algorithms are not abstract academic concepts—they are **critical mechanisms** enabling efficient memory utilization, process isolation, and system performance in every operating system. Mastering their implementation will deepen your understanding of how operating systems achieve efficiency and reliability at the memory management level.

---

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 OS-level development such as:

- **Implement** contiguous memory allocation algorithms (First-Fit, Best-Fit, Worst-Fit).
- **Manage** free and allocated memory blocks using data structures.
- **Develop** memory coalescing to merge adjacent free blocks.
- **Calculate** memory fragmentation and utilization metrics.

- **Compare** different allocation strategies and their trade-offs.
- **Handle** memory block splitting when allocation is smaller than block size.
- **Work** with linked memory block structures for efficient management.
- **Apply** algorithms for memory allocation and deallocation operations.
- **Optimize** memory usage to reduce external fragmentation.
- **Use** modular design principles for maintainable and reusable code.
- **Produce** professional-quality C code following industry standards.

4. 🎯 Learning Objectives

- **Understand** and implement contiguous memory allocation strategies used in real operating systems.
- **Apply** First-Fit algorithm for fast memory allocation with potential early fragmentation.
- **Implement** Best-Fit algorithm to minimize wasted space in allocations.
- **Design** Worst-Fit algorithm to leave larger free blocks after allocation.
- **Build** memory coalescing to merge adjacent free blocks and reduce fragmentation.
- **Calculate** fragmentation percentages and memory utilization metrics.
- **Compare** trade-offs between different allocation strategies.
- **Handle** edge cases including allocation failures and memory exhaustion.

---

5. ⚙️ Assignment Tasks

Contiguous Memory Allocation Implementation Requirements

You will implement **four core functions** using a provided modular framework:

1). First-Fit Algorithm (`first_fit.c`):

Allocate the first available memory block that is large enough

- **Key Characteristics**: Fast and simple, scans from beginning, may fragment early memory
- **Implementation Focus**: Return first suitable free block
- **Usage in OS**: Quick allocation for simple systems, embedded systems
- **Critical Aspect**: Simplest strategy but may lead to fragmentation at start of memory

2). Best-Fit Algorithm (`best_fit.c`):

Find the smallest free block that fits the allocation request

- **Key Characteristics**: Minimizes wasted space, searches entire list, may create tiny fragments
- **Implementation Focus**: Find smallest sufficient block across all blocks
- **Usage in OS**: Space-conscious systems, embedded devices with limited memory
- **Critical Aspect**: May increase external fragmentation

3). Worst-Fit Algorithm (`worst_fit.c`):

Select the largest free block for allocation

- **Key Characteristics**: Leaves larger free regions, may deplete large blocks prematurely
- **Implementation Focus**: Find largest sufficient block across all blocks
- **Usage in OS**: Systems requiring large contiguous blocks for future allocations
- **Critical Aspect**: Attempts to keep larger free blocks available but may waste memory

4). Memory Coalescing (`coalesce.c`):

Merge adjacent free memory blocks to reduce fragmentation

- **Key Characteristics**: Combines adjacent free blocks, reduces external fragmentation, enables larger allocations
- **Implementation Focus**: Detect and merge adjacent free blocks after deallocation
- **Usage in OS**: Called after process termination, critical for long-running systems
- **Critical Aspect**: Essential for maintaining memory availability and reducing fragmentation

Input/Output Specifications

The provided driver program reads commands from `INPUT.txt` and executes memory operations with your implementations.

Input File Format (`INPUT.txt`):

```
1000
ALLOCATE 1 100
ALLOCATE 2 200
ALLOCATE 3 150
STATUS
TERMINATE 2
STATUS
ALLOCATE 4 180
STATUS
```

Format Description:

- **Line 1**: Total memory size in KB
- **Subsequent lines**: Commands
  - `ALLOCATE <process_id> <size>` - Allocate memory to process
  - `TERMINATE <process_id>` - Terminate process and free memory
  - `STATUS` - Display current memory state

Expected Output Format:

```
MEMORY_SIZE: 1000
STRATEGY: FIRST_FIT

ALLOCATE: P1 100
```

```
RESULT: SUCCESS
BLOCK: P1 [0-99] SIZE:100

ALLOCATE: P2 200
RESULT: SUCCESS
BLOCK: P2 [100-299] SIZE:200

STATUS:
MEMORY_BLOCKS: 3
BLOCK: P1 [0-99] SIZE:100 STATE:ALLOCATED
BLOCK: P2 [100-299] SIZE:200 STATE:ALLOCATED
BLOCK: FREE [300-999] SIZE:700 STATE:FREE
TOTAL_FREE: 700
TOTAL_ALLOCATED: 300
FRAGMENTATION: 0.00%

TERMINATE: P2
RESULT: SUCCESS
COALESCING: 0 blocks merged

STATUS:
MEMORY_BLOCKS: 3
BLOCK: P1 [0-99] SIZE:100 STATE:ALLOCATED
BLOCK: FREE [100-299] SIZE:200 STATE:FREE
BLOCK: FREE [300-999] SIZE:700 STATE:FREE
TOTAL_FREE: 900
TOTAL_ALLOCATED: 100
FRAGMENTATION: 22.22%
```

---

6. 🗂️ Project Framework

This assignment provides a **complete development framework** including **skeleton files**, **driver program**, **Makefile**, **test cases at three difficulty levels**, and **autograder scripts** to guide your implementation process.

1) Provided Files Structure

```
PROVIDED_FILES/
├── Framework/
│   ├── memory_allocator.h          # Header with structures & prototypes
│   ├── driver.c                    # Driver (handles all I/O & commands)
│   ├── Makefile                    # Build configuration
│   ├── autograder_memory.sh        # Single submission autograder
│   ├── batchgrader_memory.sh       # Batch grading script
│   ├── TESTCASES.txt               # Default test input
│   └── EXPECTED_OUTPUT.txt         # Default expected output
│
├── Sample_Executable/
│   └── BONUS_sample.exe            # Reference implementation
│
├── Skeleton_Codes/
│   ├── first_fit_skeleton.c
```

```
16 |   ├── best_fit_skeleton.c
17 |   ├── worst_fit_skeleton.c
18 |   └── coalesce_skeleton.c
19 |
20 ├── Testcases/
21 |   ├── testcases_simple.txt            # Simple (basic allocation/deallocation)
22 |   ├── testcases_moderate.txt          # Moderate (fragmentation scenarios)
23 |   └── testcases_rigorous.txt          # Rigorous (complex memory patterns)
24 ├── BONUS_Assignment_Specification.pdf  # Detailed specification & requirement document
25 └── README.txt                          # You must submit team details in the same format
```

2) Framework Files (🔒 Do NOT modify these files)

- `memory_allocator.h`: Complete header file with data structures, constants, and function prototypes
- `driver.c`: Main program that orchestrates memory operations and manages command processing
- `Makefile`: Professional build configuration with strict compiler flags (`-std=c17 -Wall -Wextra -Werror -g -O0`)
- `autograder_memory.sh`: Single-submission automated testing script (identical to final grading mechanism)
- `batchgrader_memory.sh`: Batch grading script for processing multiple student submissions

3) Student Implementation Files (✅ You must implement these)

- `first_fit_skeleton.c` → Rename to `first_fit.c` after implementation
- `best_fit_skeleton.c` → Rename to `best_fit.c` after implementation
- `worst_fit_skeleton.c` → Rename to `worst_fit.c` after implementation
- `coalesce_skeleton.c` → Rename to `coalesce.c` after implementation

4) Sample Executable

- `BONUS_sample`: Reference implementation showing expected behavior
- Run this to understand correct output format and algorithm behavior
- Compare your output against this reference for validation

5) Testing Infrastructure

- `TESTCASES.txt`: # Test cases (**copy simple/moderate/rigorous testcases here**).
- `EXPECTED_OUTPUT.txt`: # Expected results (generated by executing **./BONUS_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 three test cases** !
  - `testcases/testcases_simple.txt` : Simple (basic allocation/deallocation)
  - `testcases/testcases_moderate.txt` : Moderate (fragmentation scenarios)
  - `testcases/testcases_rigorous.txt` : Rigorous (complex memory patterns)

⚠️ **CRITICAL REQUIREMENT**: Your implementation **MUST pass ALL three test case levels** with 100% accuracy before submission.

---

7. 🔧 Development & Testing

Recommended Implementation Order

Follow this sequence for optimal development progression:

Step 1: Setup and Understand the Framework

```
# Extract the provided files
cd PROVIDED_FILES/Framework/

# Examine the framework structure
cat memory_allocator.h        # Understand data structures
cat driver.c                  # See how driver calls your functions
cat testcases*.txt            # Study input format
cat expected_output*.txt      # Study output format

# Run sample executable to see expected behavior
cd ../Sample_Executable/
./BONUS_sample
```

Step 2: Implement Functions in Order

1. **Start with** `first_fit.c` (Simplest Algorithm)

   ```
   # Copy skeleton to working file
   cp ../Skeleton_Codes/first_fit_skeleton.c first_fit.c

   # Implement First-Fit algorithm
   # Simple linear search for first suitable block
   ```

2. **Then implement** `best_fit.c` (Minimize Waste)

   ```
   cp ../Skeleton_Codes/best_fit_skeleton.c best_fit.c

   # Implement Best-Fit algorithm
   # Find smallest sufficient block
   ```

3. **Next implement** `worst_fit.c` (Leave Large Blocks)

   ```
   cp ../Skeleton_Codes/worst_fit_skeleton.c worst_fit.c

   # Implement Worst-Fit algorithm
   # Find largest sufficient block
   ```

4. **Finally implement** `coalesce.c` (Merge Free Blocks)

```
cp ../Skeleton_Codes/coalesce_skeleton.c coalesce.c

# Implement memory coalescing
# Merge adjacent free blocks
```

Step 3: Compile and Test Incrementally

```
# Test with Level-1 (Simple) test cases

# Compile with strict flags
make rebuild

# Generate expected output from the given sample executable
./BONUS_sample > EXPECTED_OUTPUT.txt

# Copy the simple testcases in testcase file
cp Testcases/testcases_simple.txt TESTCASES.txt

# Run your implementation to generate the output
./memory_allocator  > STUDENT_OUTPUT.txt

# Compare outputs
diff STUDENT_OUTPUT.txt EXPECTED_OUTPUT.txt

# Repeat the above steps to test with MODERATE and RIGOROUS testcases
```

Step 4: Use Autograders for Final Validation

```
# Make autograders executable (if needed)
chmod +x autograder_memory.sh
chmod +x batchgrader_memory.sh

# Run the autograder (tests single submission)
./autograder_memory.sh

# Run the batchgrader (tests single submission)
# ⚠️ You must have your submission ZIP file in the current directory
./batchgrader_memory.sh
```

⚠️ **Important**: **The instructor will use the same autograder for 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**: `./memory_allocator > STUDENT_OUTPUT.txt`
3. **Compares outputs**: Extracts each strategy's output 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 for each algorithm

What Gets Compared:

- **Allocation Results**: `RESULT: SUCCESS` vs `RESULT: FAILURE`
- **Block Assignments**: Memory address ranges and sizes
- **Memory Status**: Block lists, free/allocated totals, fragmentation
- **Coalescing Operations**: Number of blocks merged

Understanding Autograder Output

When the autograder detects mismatches, it shows:

```
✅ First-Fit Algorithm: 100% match (20/20 points)

❌ Best-Fit Algorithm: 80% match (16/20 points)
   Expected Output:
      BLOCK: P2 [150-349] SIZE:200
   Your Output:
      BLOCK: P2 [100-299] SIZE:200  ← Wrong block selected

❌ Memory Coalescing: 75% match (22/30 points)
   Expected: COALESCING: 2 blocks merged
   Your Output: COALESCING: 1 blocks merged  ← Incomplete merging
```

---

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: `BONUS_FirstName_LastName.zip` (e.g., **BONUS_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 (`ZIP file`) Structure (⚠️ exact structure required - no additional files / subfolders)

> ⚠️ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: first_fit.c, best_fit.c, worst_fit.c, coalesce.c, README.txt**
>
> 📁 All other required framework files and the test cases will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as **a single ZIP archive file** to the Canvas.
- ⚠️ **Filename Format**: your `BONUS_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are 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., `BONUS_Harry_Potter.zip`). **For solo submissions** – Use your own first and last name.

```
BONUS_FirstName_LastName.zip
├── first_fit.c         # First-Fit algorithm implementation
├── best_fit.c          # Best-Fit algorithm implementation
├── worst_fit.c         # Worst-Fit algorithm implementation
├── coalesce.c          # Memory coalescing 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. `First-Fit Algorithm (first_fit.c)` implementation	20
2. `Best-Fit Algorithm (best_fit.c)` implementation	20
3. `Worst-Fit Algorithm (worst_fit.c)` implementation	20
4. `Memory Coalescing (coalesce.c)` implementation	30
5. Manual Grading - Clear code structure, Meaningful comments, well-written README, Overall readability	10
Total	**100**

⚠️ **Note**: The autograder awards 90 points automatically based on correctness. The remaining 10 points are manually graded for code quality, comments, structure, and README completeness.

🚀 ***Tips for Success:***

- Run the sample executable (`BONUS_sample`) to understand expected behavior
- Implement functions in recommended order: first_fit → best_fit → worst_fit → coalesce
- Test incrementally with all three test levels (Level-1, Level-2, Level-3)
- Ensure **exact output matching** including formatting
- Verify each algorithm selects correct blocks
- Test coalescing thoroughly with adjacent free blocks
- Handle edge cases: allocation failures, full memory, complete fragmentation
- Use autograder for validation before submission
- Maintain professional code quality throughout development

Good luck! 💪
