Assignment-3

Operating Systems

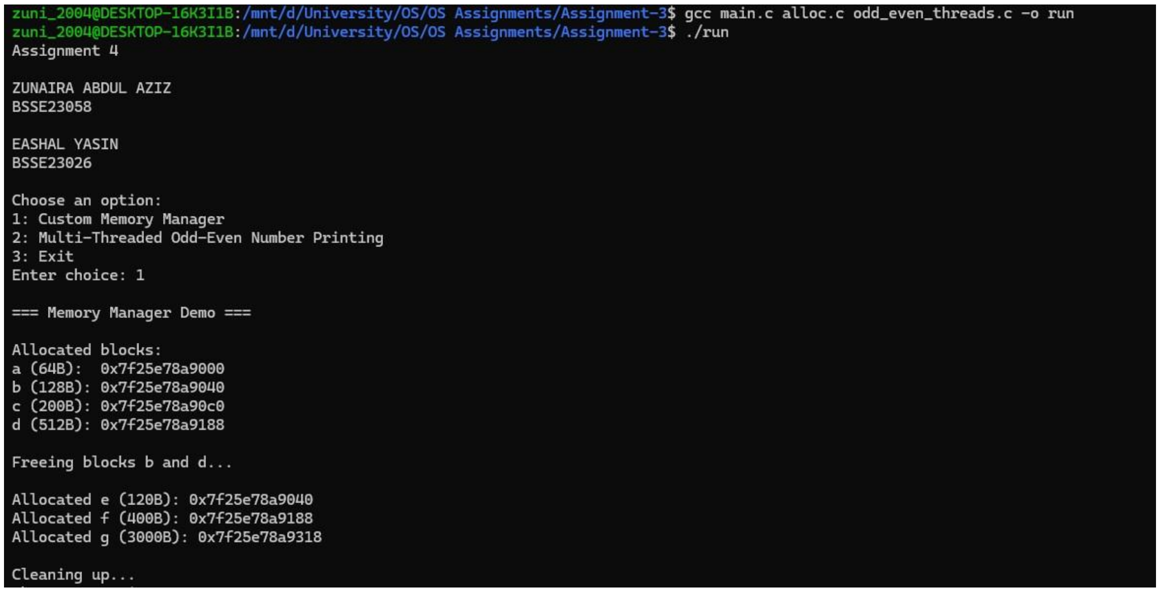
Zunaira Abdul Aziz
BSSE23058
Eashal Yasin
BSSE23026

BSSE23058 Zunaira Abdul Aziz
BSSE23026 Eashal Yasin

Part 1: Custom memory manager

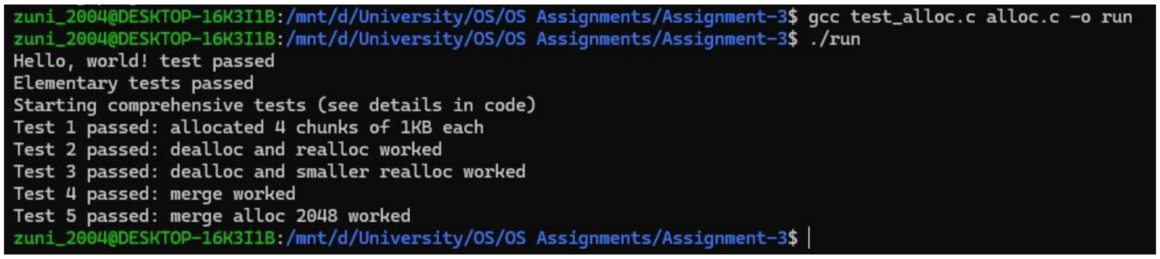
Code Files:

- **alloc.h:** Contains the function declarations for memory allocation and struct
- **alloc.c:** Actual Function Implementations
- **main.c:** menu-driven interface that allows the user to interact with the memory manager.



```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ gcc main.c alloc.c odd_even_threads.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ ./run
Assignment 4

ZUNAIRA ABDUL AZIZ
BSSE23058

EASHAL YASIN
BSSE23026

Choose an option:
1: Custom Memory Manager
2: Multi-Threaded Odd-Even Number Printing
3: Exit
Enter choice: 1

=== Memory Manager Demo ===

Allocated blocks:
a (64B):  0x7f25e78a9000
b (128B): 0x7f25e78a9040
c (200B): 0x7f25e78a90c0
d (512B): 0x7f25e78a9188

Freeing blocks b and d...

Allocated e (120B): 0x7f25e78a9040
Allocated f (400B): 0x7f25e78a9188
Allocated g (3000B): 0x7f25e78a9318

Cleaning up...
```

- **test_alloc.c:** Testing the alloc functions by test file



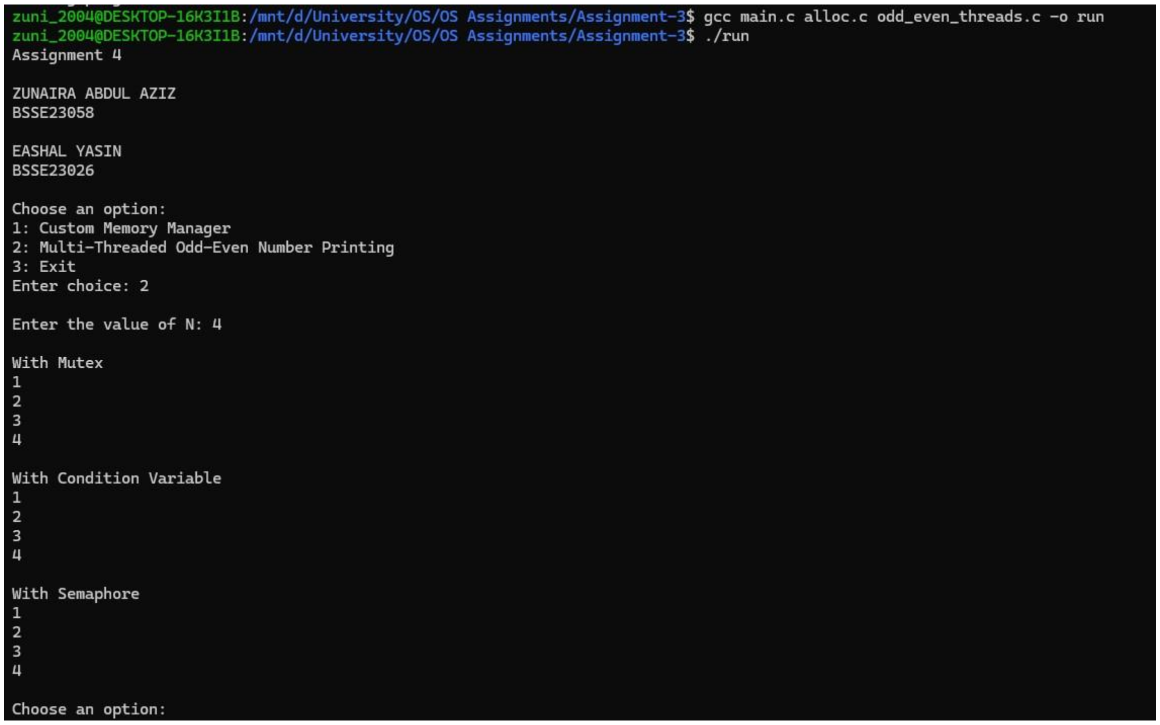
```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ gcc test_alloc.c alloc.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ ./run
Hello, world! test passed
Elementary tests passed
Starting comprehensive tests (see details in code)
Test 1 passed: allocated 4 chunks of 1KB each
Test 2 passed: dealloc and realloc worked
Test 3 passed: dealloc and smaller realloc worked
Test 4 passed: merge worked
Test 5 passed: merge alloc 2048 worked
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ |
```

Part 2: Multi-threaded even-odd number printing

Code Files:

- **odd_even_threads.h:** Contains the function declarations and struct
- **odd_even_threads.c:** Actual Function Implementations

- **main.c:** menu-driven interface that allows the user to enter the desired number of Numbers to be printed



```
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ gcc main.c alloc.c odd_even_threads.c -o run
zuni_2004@DESKTOP-16K3I1B:/mnt/d/University/OS/OS Assignments/Assignment-3$ ./run
Assignment 4

ZUNAIRA ABDUL AZIZ
BSSE23058

EASHAL YASIN
BSSE23026

Choose an option:
1: Custom Memory Manager
2: Multi-Threaded Odd-Even Number Printing
3: Exit
Enter choice: 2

Enter the value of N: 4

With Mutex
1
2
3
4

With Condition Variable
1
2
3
4

With Semaphore
1
2
3
4

Choose an option:
```

a. Which Implementation Uses the CPU More Efficiently, and Why?

The semaphore-based implementation is the most CPU-efficient among the three approaches. This is because semaphores allow threads to sleep while waiting for their turn, waking up only when explicitly signaled. The odd thread waits on odd_semaphores, and the even thread waits on even_semaphores, ensuring that only the correct thread runs at the right time. This minimizes unnecessary CPU cycles spent on lock contention or repeated condition checks.

In contrast, the mutex-only approach is less efficient because threads continuously acquire and release the mutex, even when it's not their turn. Although pthread_cond_signal is used, threads still need to recheck conditions in a loop, leading to potential busy-waiting and wasted CPU time.

The condition variable implementation improves upon the mutex-only version by allowing threads to sleep while waiting for a signal. However, it still requires frequent mutex locking and condition rechecks due to spurious wakeups, making it slightly less efficient than semaphores.

b. Which Implementation Is the Easiest to Understand and Debug?

The semaphore implementation is the most straightforward for this specific problem. The logic directly mirrors the desired behavior: the odd thread waits on odd_semaphores, prints, and signals even_semaphores, while the even thread does the opposite. This clear turn-taking mechanism makes the flow easy to follow and debug.

The mutex-only version is harder to reason about because it relies on manual condition checks inside a critical section. Without explicit waiting mechanisms, it's less intuitive to ensure strict alternation between threads.

The condition variable approach, while powerful, introduces additional complexity. The need for while loops to handle spurious wakeups and proper pairing of pthread_cond_wait with mutexes can make the code harder to understand, especially for beginners.

c. Is There Any Risk of Deadlock? How Was Safety Ensured?

All three implementations are safe from deadlock in this scenario, but each has different safety mechanisms:

- Mutex Locks: Since only one mutex is used, there's no risk of circular waits (a common deadlock cause). The lock ensures exclusive access to current, preventing race conditions.
- Condition Variables: The single mutex prevents deadlock, and the while loop ensures threads only proceed when the correct condition is met. However, incorrect signaling could lead to missed wakeups or indefinite blocking.
- Semaphores:The strict alternation between odd_semaphores and
- even_semaphores ensures that threads only proceed when signaled, eliminating deadlock risk. If semaphores are initialized and posted correctly, the flow remains deadlock-free.

Safety was ensured by:

- Properly initializing synchronization primitives (mutexes, condition variables, semaphores).
- Using while loops for condition checks (preventing spurious wakeups).
- Ensuring every wait has a corresponding signal/post.

Conclusion

- Best for CPU efficiency: Semaphores (threads sleep until needed).
- Best for readability: Semaphores (clear signaling logic).

- Safest: All are safe if used correctly, but semaphores provide the most structured approach.