**Name: Mayesha Bintha Mizan**
**ID-811281302**

The goal of this project is to investigate, identify, and resolve bugs with minimal changes to the existing codebase of four different applications.

## Worker – Segmentation Fault

This is a simple toy program that simulates a multi-threaded worker system. The system experiences a segmentation fault under certain conditions, particularly during shutdown when multiple threads interact with shared data.

### 1. Problem Identification:

The segmentation fault occurs when the **main thread attempts to access shared data** after receiving a SIGINT signal, leading to a concurrency issue, specifically a race condition. This problem arises at line 51, where multiple threads simultaneously update the global thread_data variable **without proper synchronization mechanisms** in place, causing undefined behavior and ultimately resulting in a segmentation fault.

### 2. Bug Reproduction:

I Run the program (./worker) and send a SIGINT (ctrl+c) multiple times to reproduce the segmentation fault. I used a **bash script** to automate the process and run the program multiple times to consistently reproduce the bug:

```
# Bash script to run the program repeatedly
for i in {1..1000}
do
    ./worker
    if [ $? -ne 0 ]; then
        break
    fi
done
```

After several runs, the segmentation fault observed.

```
^CMain: caught SIGINT (Ctrl+C). Exiting gracefully.
Thread #9: terminating..
Thread #5: terminating..
Thread #0: terminating..
Thread #7: terminating..
Thread #2: terminating..
Thread #1: terminating..
Thread #3: terminating..
Thread #8: terminating..
Thread #6: terminating..
Thread #4: terminating..
total = 5478
^CMain: caught SIGINT (Ctrl+C). Exiting gracefully.
^CMain: caught SIGINT (Ctrl+C). Exiting gracefully.
^CSegmentation fault (core dumped)
[mb69711@csci-odin worker]$ ^C
[mb69711@csci-odin worker]$
```

**Name: Mayesha Bintha Mizan**
**ID-811281302**

**3. Root Cause Analysis:**

The issue on line 51 of the original code involved accessing the shared data (thread_data) without proper synchronization, leading to a race condition. Both the main thread and worker threads were concurrently modifying and accessing this data, there **is no mutual exclusion** for accessing which led to unsafe memory access and ultimately caused the segmentation fault. During shutdown, the main thread tried to free the shared data while worker threads were still using it, resulting in an inconsistent or partially modified state and causing instability and crashes.

**4.Fix Design:**

To fix the worker program, I introduced a **mutex (pthread_mutex_lock)** to synchronize access to the shared data (thread_data). I wrapped all modifications and reads of the shared data with **pthread_mutex_lock(&lock) and pthread_mutex_unlock(&lock),** ensuring that only one thread could access or modify the data at any given time. This effectively eliminated the race condition by ensuring that the main thread would not access deallocated or partially modified data. The mutex lock provided mutual exclusion, allowing each thread to safely perform its operations on the shared data without interference.

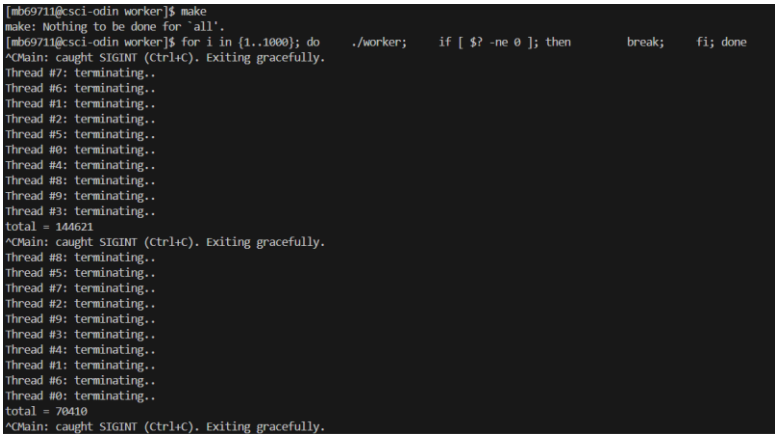
**5. Discussion of Fixes:**

The fix applied to the worker program was effective because it addressed the core concurrency issue: improper synchronization and overlapping access. By using a mutex lock, I was able to ensure that all access to shared data was properly coordinated, preventing race conditions and segmentation faults. The minimal changes applied adding synchronization primitives ensured that the program's structure remained largely intact while providing thread safety.

The use of the mutex prevented any thread from accessing the data while another was modifying or freeing it, effectively eliminating the race condition that led to the segmentation fault. It required minimal changes to the code while providing a significant improvement in stability.

**6. Testing and Validation:**

I re-ran the program using the same bash script after adding the mutex to verify that the segmentation fault no longer occurs, even after multiple Ctrl+C signals. The program now exit gracefully every time, with all threads properly terminating.



**Name: Mayesha Bintha Mizan**
**ID-811281302**

## 2. StringBuffer – Assertion Failed

The StringBuffer class is a thread-safe utility that provides operations such as append, erase, insert, and print for managing strings. Despite using mutex locks (mutex_lock) to protect access to the shared string buffer, the program experienced an assertion failure.

**1. Problem Identification:**

Here I observe concurrency issues in the StringBuffer implementation. The StringBuffer implementation had an **assertion failure** when **multiple threads attempted to read and write concurrently** without proper synchronization. This led to **inconsistent states** within the buffer, especially during multi-step operations such as append and erase, which were not handled atomically.

**2. Bug Reproduction:**

When I run the test case I can produce the bug. This test exposed the assertion failure when both threads accessed the shared buffer simultaneously, leading to inconsistent data states.

```
[mb69711@csci-odin stringbuffer]$ ./test
0th try - Thread 1: erase and append: 0
0th try - Thread 2: create and append: 0
100th try - Thread 2: create and append: 1
100th try - Thread 1: erase and append: 1
200th try - Thread 1: erase and append: 2
200th try - Thread 2: create and append: 2
300th try - Thread 2: create and append: 3
300th try - Thread 1: erase and append: 3
400th try - Thread 2: create and append: 4
400th try - Thread 1: erase and append: 4
500th try - Thread 2: create and append: 5
500th try - Thread 1: erase and append: 5
600th try - Thread 2: create and append: 6
600th try - Thread 1: erase and append: 6
700th try - Thread 2: create and append: 7
700th try - Thread 1: erase and append: 7
800th try - Thread 2: create and append: 8
800th try - Thread 1: erase and append: 8
900th try - Thread 2: create and append: 9
900th try - Thread 1: erase and append: 9
1000th try - Thread 2: create and append: 10
1000th try - Thread 1: erase and append: 10
test: stringbuffer.cpp:50: void StringBuffer::getChars(int, int, char*, int): Assertion `0' failed.
Aborted (core dumped)
[mb69711@csci-odin stringbuffer]$ ./test
```

**3. Root Cause Analysis:**

The root cause of the assertion failure in the StringBuffer class was an incorrect usage pattern of the mutex. Although mutexes were being used, the locking strategy did not account for overlapping operations that could affect the internal state of the buffer. This led to situations where a thread was modifying the buffer while another thread was reading from or erasing parts of it, causing inconsistent state and triggering the assertion failure. The methods in the StringBuffer class were individually protected by mutexes, but there were gaps in ensuring that sequences of operations involving multiple steps were atomic.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

**4. Fix Design:**

For the StringBuffer implementation, I introduced a **read-write lock** to synchronize concurrent access to the buffer. Read-Write Locks allow multiple threads to read simultaneously but require exclusive access for writing. Write operations, such as erasing and appending, were protected with a write lock (pthread_rwlock), while read operations were protected with a read lock (pthread_rdlock). This allowed multiple threads to read concurrently while ensuring that write operations had exclusive access, preventing any inconsistencies.

**5. Discussion of Fixes:**

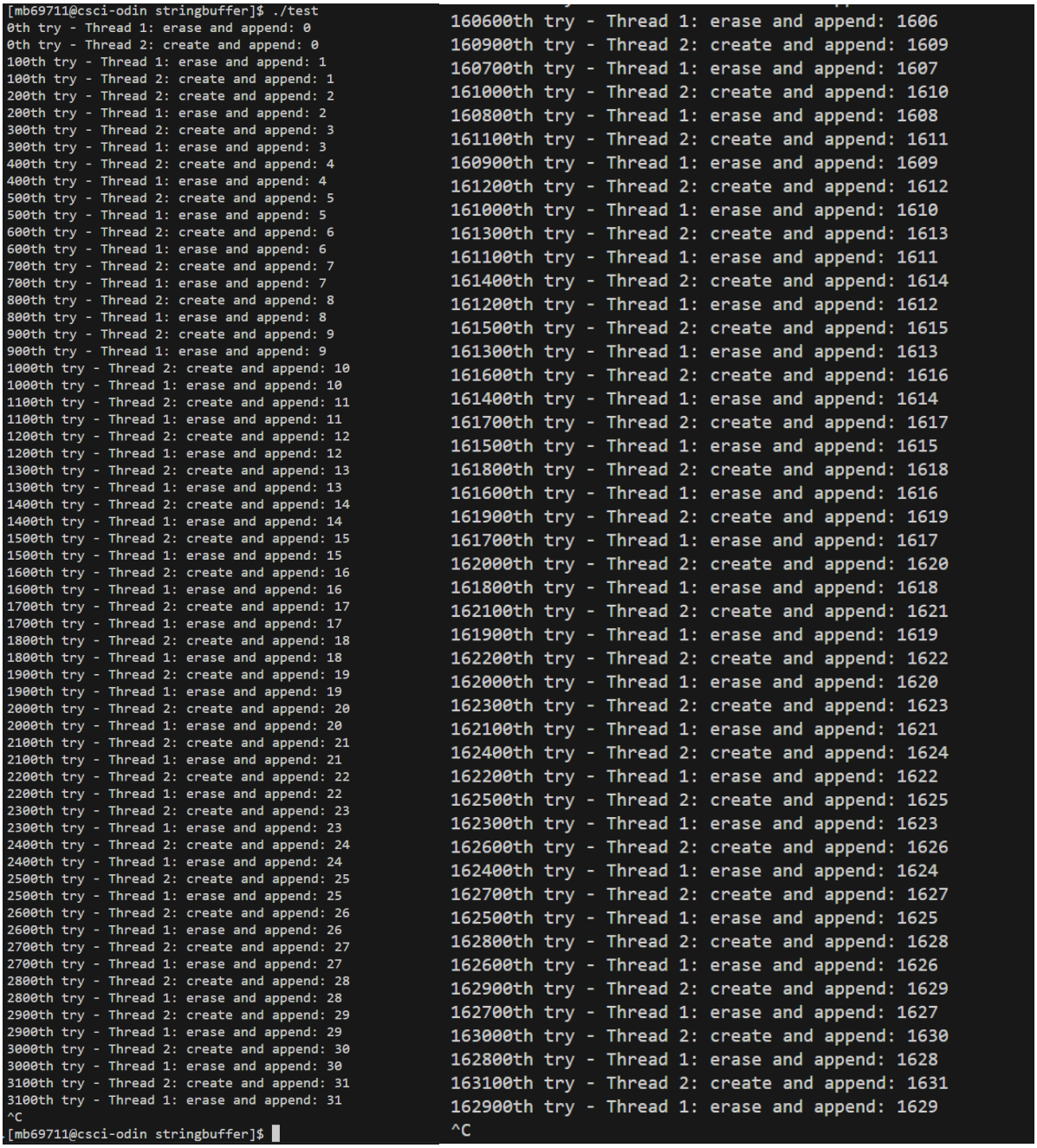
The fix applied to the StringBuffer implementation was effective because it addressed the core concurrency issue: improper synchronization and overlapping access. By using read-write locks, I was able to ensure that all access to shared data was properly coordinated, preventing race conditions and assertion failures. The minimal changes on 2 threads applied adding synchronization primitives ensured that the program's structure remained largely intact while providing thread safety.

However, there are trade-offs to these fixes. The use of mutex locks introduces a slight performance overhead due to the added locking and unlocking operations, which can impact performance in highly concurrent environments. Similarly, read-write locks can lead to writer starvation if there are many read operations, although this was not observed during testing. Despite these trade-offs, the fixes are worth it to ensure data integrity and prevent crashes.

**6. Testing and Validation:**

For the StringBuffer implementation, I ran the modified test.cpp program with two threads performing concurrent operations on the shared buffer. The program ran consistently without any assertion failures, and the output showed that the operations were executed in an orderly manner.

**Name: Mayesha Bintha Mizan**
**ID-811281302**




A screenshot of the output from the test program showed thousands of successful operations performed by both threads without any inconsistencies, demonstrating that the read-write lock was effective in preventing race conditions. Logs confirmed that no new bugs were introduced and that the fixes provided the necessary thread safety.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

## 3. Pbzip2

PBZIP2 is a parallel version of bzip2, designed to utilize multiple CPU cores for faster compression and decompression of large files. By dividing the compression workload into smaller, independent chunks, PBZIP2 significantly speeds up the processing of large datasets.

**1. Problem Identification:**

While working on pbzip2, a parallel version of the bzip2 compression tool, an issue was observed that caused the program to crash during execution with a segmentation fault. The segmentation fault appeared intermittently, suggesting that it might be related to threading issues, such as improper synchronization, or order violations.

The issue occurred when running the compression process using multiple threads, where one of the threads encountered an error when attempting to unlock a mutex.Then I found that the mutex had become **NULL** unexpectedly, leading to a crash when attempting to unlock it. This **null state** of the mutex pointed to an improper handling or premature release of the mutex, resulting in the program crashing due to attempting to unlock a NULL mutex. This indicates a deeper synchronization issue, where the mutex was not being properly managed across the different threads, potentially causing undefined behavior and segmentation faults.

**Additionally,** another bug was identified related to the improper setting of the allDone flag, which led to unexpected program behavior. The allDone variable is shared across multiple threads (producers, consumers, and file writer). There is no mutex or other synchronization mechanism to protect access to allDone.

**2. Bug Reproduction**

To reproduce the bug, I added artificial delays before critical lines of code to expose potential race conditions and order violations that were not consistently triggered in normal operation. The delay was added before the last line in a critical section where a mutex was being unlocked, as well as in other one place to delay the exit of the main program. These delays were necessary because the program ran too quickly in the Odin environment, making it difficult to reproduce the bug without slowing down thread execution. By adding these delays, the system was forced into a state where the bug became more apparent, ultimately resulting in a consistent segmentation fault in the consumer thread. Debugging with gdb confirmed that the segmentation fault occurred due to an improper unlock attempt on a mutex, supporting the hypothesis of an order violation in thread synchronization.

To reproduce the issue related to the allDone flag, a thorough examination of the program flow was conducted to identify cases where allDone was set prematurely or incorrectly. By adding logging statements and monitoring the flag's status across different threads, it became evident that improper handling of allDone led to threads exiting too early, causing incomplete processing of data blocks.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

```
[mb69711@csci-odin pbzip2]$ ./pbzip2 -k -f test.file
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

         # CPUs: 36
 BWT Block Size: 900k
File Block Size: 900k
----------------------------------------
         File #: 1 of 1
     Input Name: test.file
    Output Name: test.file.bz2

     Input Size: 2930969 bytes
Compressing data...
    Output Size: 875176 bytes
----------------------------------------

     Wall Clock: 3.712306 seconds
Segmentation fault (core dumped)
[mb69711@csci-odin pbzip2]$ gdb pbzip2
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/myid/mb69711/OS_Proj2/pbzip2/pbzip2...(no debugging symbols found)...don
(gdb) run -k -f test.file
Starting program: /home/myid/mb69711/OS_Proj2/pbzip2/pbzip2 -k -f test.file
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

         # CPUs: 36
 BWT Block Size: 900k
File Block Size: 900k
```

```
----------------------------------------
        File #: 1 of 1
     Input Name: test.file
    Output Name: test.file.bz2

     Input Size: 2930969 bytes
Compressing data...
[New Thread 0x7ffff6dc0700 (LWP 3899)]
[New Thread 0x7ffff65bf700 (LWP 3900)]
[New Thread 0x7ffff5dbe700 (LWP 3901)]
[New Thread 0x7ffff55bd700 (LWP 3902)]
[New Thread 0x7ffff4dbc700 (LWP 3903)]
[New Thread 0x7ffff45bb700 (LWP 3904)]
[New Thread 0x7ffff3dba700 (LWP 3905)]
[New Thread 0x7ffff35b9700 (LWP 3906)]
[New Thread 0x7ffff2db8700 (LWP 3907)]
[New Thread 0x7ffff25b7700 (LWP 3908)]
[New Thread 0x7ffff1db6700 (LWP 3911)]
[New Thread 0x7ffff15b5700 (LWP 3912)]
[New Thread 0x7ffff0db4700 (LWP 3914)]
[New Thread 0x7ffff05b3700 (LWP 3915)]
[New Thread 0x7fffefdb2700 (LWP 3916)]
[New Thread 0x7fffef5b1700 (LWP 3917)]
[New Thread 0x7fffeedb0700 (LWP 3918)]
[New Thread 0x7fffee5af700 (LWP 3919)]
[New Thread 0x7fffeddae700 (LWP 3920)]
[New Thread 0x7fffed5ad700 (LWP 3921)]
[New Thread 0x7fffecdac700 (LWP 3922)]
[New Thread 0x7fffec5ab700 (LWP 3923)]
[New Thread 0x7fffebdaa700 (LWP 3924)]
[New Thread 0x7fffeb5a9700 (LWP 3925)]
[New Thread 0x7fffeada8700 (LWP 3926)]
[New Thread 0x7fffea5a7700 (LWP 3927)]
[New Thread 0x7fffe9da6700 (LWP 3928)]
[New Thread 0x7fffe95a5700 (LWP 3929)]
[New Thread 0x7fffe8da4700 (LWP 3930)]
[New Thread 0x7fffe85a3700 (LWP 3931)]
[New Thread 0x7fffe7da2700 (LWP 3932)]
[New Thread 0x7fffe75a1700 (LWP 3933)]
[New Thread 0x7fffe6da0700 (LWP 3934)]
[New Thread 0x7fffe659f700 (LWP 3935)]
[New Thread 0x7fffe5d9e700 (LWP 3936)]
[New Thread 0x7fffe559d700 (LWP 3937)]
[New Thread 0x7fffe4d9c700 (LWP 3938)]
    Output Size: 875176 bytes
[Thread 0x7fffe4d9c700 (LWP 3938) exited]
----------------------------------------

     Wall Clock: 3.697705 seconds

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff3dba700 (LWP 3905)]
0x00007ffff7bc9ee0 in pthread_mutex_unlock () from /lib64/libpthread.so.0
Missing separate debuginfos, use: debuginfo-install bzip2-libs-1.0.6-13.el7.x86_64 glibc-2.17-326.el7_9.3.x86_64 libgcc-4.8.5-44.el7.x86_64 libstdc++-4.8.5-44.el7.x86_64
(gdb) bt
#0  0x00007ffff7bc9ee0 in pthread_mutex_unlock () from /lib64/libpthread.so.0
#1  0x0000000000402c01 in consumer(void*) ()
#2  0x00007ffff7bc6ea5 in start_thread () from /lib64/libpthread.so.0
#3  0x00007ffff6ebfb0d in clone () from /lib64/libc.so.6
(gdb)
```

## 3. Root Cause Analysis:

**Name: Mayesha Bintha Mizan**
**ID-811281302**

The root cause of the segmentation fault was traced to an order violation in thread synchronization, specifically related to improper handling of the sequence in which mutexes were being locked and unlocked. The consumer threads were occasionally attempting to unlock a mutex that had not been properly locked or was already unlocked by another thread, leading to undefined behavior. This issue was exacerbated by the presence of multiple threads operating concurrently, which made the timing between operations critical. The lack of proper coordination between condition variables, mutexes, and shared flags led to inconsistent states, resulting in segmentation faults. By adding artificial delays, it became evident that the timing and sequence of locking and unlocking mutexes were crucial, and slight variations in thread scheduling could easily trigger the fault.

Further analysis revealed that the incorrect setting of the allDone flag, which was intended to signal the completion of data processing. However, due to improper updates to this flag, some threads prematurely exited their loops, resulting in incomplete compression or decompression of data blocks. This was particularly problematic in scenarios where multiple consumer threads were waiting for work, but the premature setting of allDone led them to believe that no more work was available.

**4. Fix Design:**

To address the identified issues, several changes were implemented to improve thread synchronization and ensure proper handling of shared resources:

1. **Consumer Threads Management:** The consumer threads were restructured to be stored in an **array**, allowing for better control and management. Instead of dynamically creating threads without a clear mechanism to join them, each consumer thread was explicitly created and stored in an array. After all the consumer threads were launched, pthread_join was used to **wait** for each consumer thread to complete before proceeding. This ensured that all threads finished their work properly and that no thread was left hanging or terminated prematurely.

```
ret = pthread_create(&con[i], NULL, consumer_decompress, fifo);
if (ret != 0)
{
        fprintf(stderr, " *ERROR: Not enough resources to create consumer thread #%d (code = %d)  Aborting...
        return 1;
}
```

2. **Synchronization with Output Thread:** The output thread (fileWriter) was also synchronized more effectively with the consumer threads. The output thread now waits for all consumer threads to **finish** processing before **writing the final output**, ensuring that no data is lost or partially written due to premature termination of consumer threads.

```
// wait until exit of thread
pthread_join(output, NULL);
for (i=0; i < numCPU; i++)
        {
                pthread_join(con[i], NULL);
        }
```

3. **allDone Flag Protection:** The allDone flag, which indicates when the producer has finished adding work to the queue, was modified to be updated in a thread-safe manner. Previously, allDone was being updated without adequate protection, which led to race conditions where consumer threads could read an incorrect value. The new design uses a mutex to protect access to the allDone flag, ensuring that updates are consistent and properly synchronized across all threads.

```
pthread_mutex_lock(&allDoneMutex);
allDone = 1;  // Signal that the producer is done
pthread_mutex_unlock(&allDoneMutex);;
return 0;
```

**4.Discussion of Fixes:**

The implemented fixes effectively addressed the concurrency issues by enhancing the synchronization mechanisms between threads. The use of an array to manage consumer threads and employing pthread_join ensured that all threads were properly joined before proceeding, thereby eliminating the possibility of threads being terminated prematurely or attempting to access shared resources in an undefined state. This fix also provided a clear structure for managing thread lifetimes, which reduced the likelihood of resource leaks or undefined behavior.

By protecting the allDone flag with a mutex, the changes ensured that all threads had a consistent view of the program's state, preventing premature exits and incomplete processing of data blocks. This fix eliminated the race conditions associated with the flag and ensured that consumer threads only exited when all work was genuinely completed.

While these fixes resolved the observed issues, there are potential trade-offs to consider. The additional synchronization mechanisms, such as protecting the allDone flag with a mutex and using pthread_join for all consumer threads, introduce slight overhead in terms of performance. However, this overhead is justified by the increased reliability and correctness of the program. The likelihood of new bugs being introduced is minimal, as the changes primarily focus on ensuring proper thread management and synchronization, which are well-established practices in concurrent programming. Nevertheless, careful testing is recommended to verify that the fixes perform well under various workloads and that no new deadlocks or performance bottlenecks are introduced.

**6. Testing and Validation:**

The fixed version of pbzip2 was compiled and executed multiple times under varying conditions to ensure stability and correctness. The screenshots provided demonstrate successful runs without segmentation faults, with all threads exiting normally and the correct output being generated. All threads exited normally, and there were no stack traces indicating a crash. This provides strong evidence that the fixes were effective in resolving the original issues.

Additionally, logging was used to verify the correct sequence of operations, including proper locking and unlocking of mutexes, correct updates to the allDone flag, and orderly thread termination. No new bugs were observed during the testing phase, confirming the stability of the fixed design.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

```
[mb69711@csci-odin pbzip2]$ make
g++ -O3 -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -o pbzip2 pbzip2.cpp -pthread -lpthread -lbz2
[mb69711@csci-odin pbzip2]$ ./pbzip2 -k -f test.file
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

         # CPUs: 36
 BWT Block Size: 900k
File Block Size: 900k
-------------------------------------------
         File #: 1 of 1
     Input Name: test.file
    Output Name: test.file.bz2

     Input Size: 2930969 bytes
Compressing data...
    Output Size: 875176 bytes
-------------------------------------------

     Wall Clock: 4.608163 seconds
[mb69711@csci-odin pbzip2]$ gdb pbzip2
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/myid/mb69711/OS_Proj2/pbzip2/pbzip2...(no debugging symbols found)...done.
(gdb) run -k -f test.file
Starting program: /home/myid/mb69711/OS_Proj2/pbzip2/pbzip2 -k -f test.file
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Parallel BZIP2 v0.9.4 - by: Jeff Gilchrist [http://compression.ca]
[Aug. 30, 2005]             (uses libbzip2  by Julian Seward)

** This is a BETA version - Use at your own risk! **

         # CPUs: 36
 BWT Block Size: 900k
File Block Size: 900k
```

```
[New Thread 0x7fffe75a1700 (LWP 23054)]
[New Thread 0x7fffe6da0700 (LWP 23055)]
[New Thread 0x7fffe659f700 (LWP 23056)]
[New Thread 0x7fffe5d9e700 (LWP 23057)]
[New Thread 0x7fffe559d700 (LWP 23058)]
[New Thread 0x7fffe4d9c700 (LWP 23059)]
[Thread 0x7ffff2db8700 (LWP 23031) exited]
    Output Size: 875176 bytes
[Thread 0x7fffe4d9c700 (LWP 23059) exited]
[Thread 0x7ffff6dc0700 (LWP 23023) exited]
[Thread 0x7ffff25b7700 (LWP 23032) exited]
[Thread 0x7ffff3dba700 (LWP 23029) exited]
[Thread 0x7ffff35b9700 (LWP 23030) exited]
[Thread 0x7ffff45bb700 (LWP 23028) exited]
[Thread 0x7ffff4dbc700 (LWP 23027) exited]
[Thread 0x7ffff55bd700 (LWP 23026) exited]
[Thread 0x7ffff5dbe700 (LWP 23025) exited]
[Thread 0x7fffe559d700 (LWP 23058) exited]
[Thread 0x7ffff15b5700 (LWP 23034) exited]
[Thread 0x7ffff1db6700 (LWP 23033) exited]
[Thread 0x7ffff65bf700 (LWP 23024) exited]
[Thread 0x7fffe5d9e700 (LWP 23057) exited]
[Thread 0x7fffe85a3700 (LWP 23052) exited]
[Thread 0x7fffee5af700 (LWP 23040) exited]
[Thread 0x7fffe659f700 (LWP 23056) exited]
[Thread 0x7fffecdac700 (LWP 23043) exited]
[Thread 0x7fffefdb2700 (LWP 23037) exited]
[Thread 0x7fffec5ab700 (LWP 23044) exited]
[Thread 0x7fffeada8700 (LWP 23047) exited]
[Thread 0x7fffeddae700 (LWP 23041) exited]
[Thread 0x7fffe95a5700 (LWP 23050) exited]
[Thread 0x7fffe6da0700 (LWP 23055) exited]
[Thread 0x7fffea5a7700 (LWP 23048) exited]
[Thread 0x7fffe75a1700 (LWP 23054) exited]
[Thread 0x7fffebdaa700 (LWP 23045) exited]
[Thread 0x7fffe7da2700 (LWP 23053) exited]
[Thread 0x7fffe8da4700 (LWP 23051) exited]
[Thread 0x7fffeb5a9700 (LWP 23046) exited]
[Thread 0x7ffff0db4700 (LWP 23035) exited]
[Thread 0x7fffe9da6700 (LWP 23049) exited]
[Thread 0x7fffef5b1700 (LWP 23038) exited]
[Thread 0x7fffed5ad700 (LWP 23042) exited]
[Thread 0x7fffeedb0700 (LWP 23039) exited]
[Thread 0x7ffff05b3700 (LWP 23036) exited]
-------------------------------------------

    Wall Clock: 4.625774 seconds
[Inferior 1 (process 23019) exited normally]
Missing separate debuginfos, use: debuginfo-install bzip2-libs-1.0.6-13.el7.x86_64 glibc-2.17-326.el7_9.3.x86_64 libgcc-4.8.5-44.el7.x86_64 libstdc++-4.8.5-44.el7.x86_64
(gdb) bt
No stack.
(gdb)
```

**Name: Mayesha Bintha Mizan**
**ID-811281302**

I also downloaded both files to my laptop and compare them and ensure that they are same file.

```
C:\Users\Mayesha Rahman>comp "C:\Users\Mayesha Rahman\Desktop\test.download" "C:\Users\Mayesha Rahman\Desktop\512MB.zip"
Comparing C:\Users\Mayesha Rahman\Desktop\test.download and C:\Users\Mayesha Rahman\Desktop\512MB.zip...
Files compare OK


Compare more files (Y/N) ?
```

## 4. Aget – Corrupted Log

Aget is a multithreaded download accelerator that splits a file into multiple segments and downloads them concurrently to increase download speed. By using multiple threads, it effectively maximizes bandwidth utilization.

### 1. Problem Identification:

During the investigation of Aget, two key bugs were identified. The first issue relates to **improper** handling of the **download progress log**, which led to corrupted log files. This issue is a race condition caused by the concurrent access and modification of the **shared log file by multiple threads** without **appropriate synchronization**. The threads involved in the download process updated the shared thread_data structure and log file simultaneously, leading to inconsistencies. This race condition resulted in a corrupted log file that failed verification, as highlighted by the verifier tool indicating that the actual bytes written were inconsistent with the expected content size.

The second issue relates to improper thread management when an interruption (e.g., SIGINT) was received during the download. This caused the main thread to improperly access shared data, potentially after one of the download threads had already modified or released it. This problem is also a race condition, exacerbated by the fact that the interruption occurred asynchronously, leaving the threads in an inconsistent state. This led to the improper termination of the download job, incomplete log updates, and inconsistent progress.

### 2. Bug Reproduction:

When I used a small video file, I couldn't get the bug to show up. Everything seemed fine. Then I choose a large file. The first time I compiled and downloaded it, everything worked smoothly.

```
[mb69711@csci-odin aget]$ make
make: Nothing to be done for `all'.
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > Couldn't find log file for this download, starting a clean job...
Aget > Head-Request Connection established
Aget > Downloading /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
^C^C caught, terminating download job. Please wait...21% done] Bytes: 115748358
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 137884606 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is okay. It is safe to resume the download.
[mb69711@csci-odin aget]$
```

The log file records that **137,884,606 bytes** have been transferred. Then I tried resuming the download. It successfully recognizes that **137,884,606 bytes** have already been transferred and I

interrupted again at **66% progress**. This time, The verifier reports that the log file was incorrect and might be corrupted, and then deleted the log.

```
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > 137884606 bytes already transferred
Aget > Resuming download /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
^C^C caught, terminating download job. Please wait...66% done] Bytes: 358019525
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 406581829 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is incorrect and may be corrupted. Deleting the log.
Total bytes written: 406581829, Size of content: 268697223 bytes.
[mb69711@csci-odin aget]$
```

With no log file left, when I tried to download the same file again, it said it couldn't find the log and started from scratch.

```
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is incorrect and may be corrupted. Deleting the log.
Total bytes written: 406581829, Size of content: 268697223 bytes.
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > Couldn't find log file for this download, starting a clean job...
Aget > Head-Request Connection established
Aget > Downloading /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
File already exists! Overwrite?(y/n) y
^C^C caught, terminating download job. Please wait...16% done] Bytes: 90631886
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 101340518 bytes have been transferred
```

**3. Root Cause Analysis:**

The core issue appears to be related to the corruption of the log file that tracks download progress. Upon analyzing the process, we found that the save_log function in Resume.c is responsible for saving the download state whenever an interruption occurs. Another potential root cause identified is related to **inconsistencies in the offsets** maintained by each thread. For each thread, the soffset **(starting offset**) value should remain **consisten**t with the value that was initially assigned when the download started. However, during resumption, the offset values are not correctly restored, which likely results in the incorrect continuation of the download, leading to log corruption.

The code uses pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL); to handle thread cancellation. Before entering a critical section, it's important to use **pthread_ setcanceltype (PTHREAD_CANCEL_DEFERRED, NULL);** to ensure that cancellation doesn't happen until the critical section is fully executed. This prevents issues where a thread is cancelled at an unsafe point, which could lead to inconsistent or corrupted states.

The download process is managed by the http_get function in Download.c, which is responsible for tracking the total bytes written. The log-saving function (save_log) is called whenever the download is interrupted. This means that if there is any flaw in the way the total bytes are tracked or saved, the resulting log will be inaccurate.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

**4. Fix Design:**

The primary fix was to ensure that **each thread's soffset (starting offset)** and **offset values** are accurately restored during the resumption of a download. In the original implementation, offset was being re-initialized, leading to discrepancies in the continuation process. The fix guarantees that the original starting offset (soffset) is consistently reapplied when threads are resumed, which helps maintain data integrity.

```
if ((dr - i ) > foffset)
        dw = pwrite(td->fd, s, (foffset - i), td->soffset);
else
        dw = pwrite(td->fd, s, (dr - i), td->soffset);
td->offset = td->soffset + dw;
pthread_mutex_lock(&bwritten_mutex);
bwritten += dw;
pthread_mutex_unlock(&bwritten_mutex);

pthread_testcancel();   /* Check for pending cancel requests */
```

```
        i++;
}
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
if ((dr - i ) > foffset)
        dw = pwrite(td->fd, s, (foffset - i), td->offset);
else
        dw = pwrite(td->fd, s, (dr - i), td->offset);
td->offset += dw;
pthread_mutex_lock(&bwritten_mutex);
bwritten += dw;
pthread_mutex_unlock(&bwritten_mutex);

pthread_testcancel();   /* Check for pending cancel requests */
```

The changes were implemented in Download.c by modifying the assignment of soffset to each thread during the retry phase. This ensured that all threads have the correct initial reference point for resuming their download segment.

The use of pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL); was adjusted by adding a **PTHREAD_CANCEL_DEFERRED** in critical section work. This change was minimal but critical in ensuring that threads are only canceled at specific, safe points, reducing the risk of log file corruption during a forced interruption.

This adjustment ensures that the save_log function in Resume.c has **sufficient time** to complete its operations, thereby preventing partially saved or corrupted log states.

**5. Discussion of Fixes:**

The implemented fixes have proven effective in addressing the key issues related to download resumption and log corruption. **Synchronizing offsets** ensure that the download progress managed by each thread remains consistent across interruptions. This consistency is crucial for preventing threads from writing over the wrong portions of data, which was one of the primary contributors to the observed log corruption. The adjustment to **PTHREAD_CANCEL_DEFERRED** further guarantees that threads are not abruptly terminated during critical operations, particularly during the saving of logs, thereby enhancing the reliability of the resumption process. Despite their effectiveness, the fixes do come with potential trade-offs. For instance, using PTHREAD_CANCEL_DEFERRED may reduce the system's responsiveness to user interruptions. Threads will now only be canceled at specific, safer points, which means a slight delay may be experienced when attempting to stop a download. This trade-off was deemed acceptable as it ensures the integrity of the log file and prevents data corruption, which is more critical for maintaining reliable download resumption functionality. By ensuring that thread offsets are correctly managed, by carefully controlling thread cancellation points, and by enhancing the consistency of log-saving operations, the fixes make the download process much more resilient to interruptions.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

**6. Testing and Validation:**

After making the necessary changes, I start download and interrupted the download at 21% completion and then resumed it. The log file worked perfectly, it recorded the progress accurately, and I was able to continue downloading from where it left off. I interrupted it again at 66%, and the same thing happened: the log saved properly, and the download was successfully completed. In total, it took 13 seconds to finish, and everything looked good.

```
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > Couldn't find log file for this download, starting a clean job...
Aget > Head-Request Connection established
Aget > Downloading /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
^C^C caught, terminating download job. Please wait. .21% done] Bytes: 115483478
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 121815478 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is okay. It is safe to resume the download.
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > 121815478 bytes already transferred
Aget > Resuming download /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
Start: 0, Finish: 268435456, Offset: 61841299, Diff: 61841299
Start: 268435456, Finish: 536870912, Offset: 328409635, Diff: 59974179
^C^C caught, terminating download job. Please wait. .62% done] Bytes: 335132477
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 341270549 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is okay. It is safe to resume the download.
[mb69711@csci-odin aget]$ ./aget -n2 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > 341270549 bytes already transferred
Aget > Resuming download /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 2
Start: 0, Finish: 268435456, Offset: 167735295, Diff: 167735295
Start: 268435456, Finish: 536870912, Offset: 441970710, Diff: 173535254
Aget > Download completed, job completed in 5 seconds. (38203 Kb/sec) 514917236
Aget > Total download time: 13 seconds. Overall speed: 40329 Kb/sec
Aget > Shutting down...
[mb69711@csci-odin aget]$
```

Next, I tested with 4 threads. Again, I interrupted the download, this time at 23%. The log file was correct, and I resumed the download without any problems. I repeated the interruption and resumption process, and everything worked as expected. The download finished in 7 seconds, showing that the fix was reliable even with more threads.

**Name: Mayesha Bintha Mizan**
**ID-811281302**

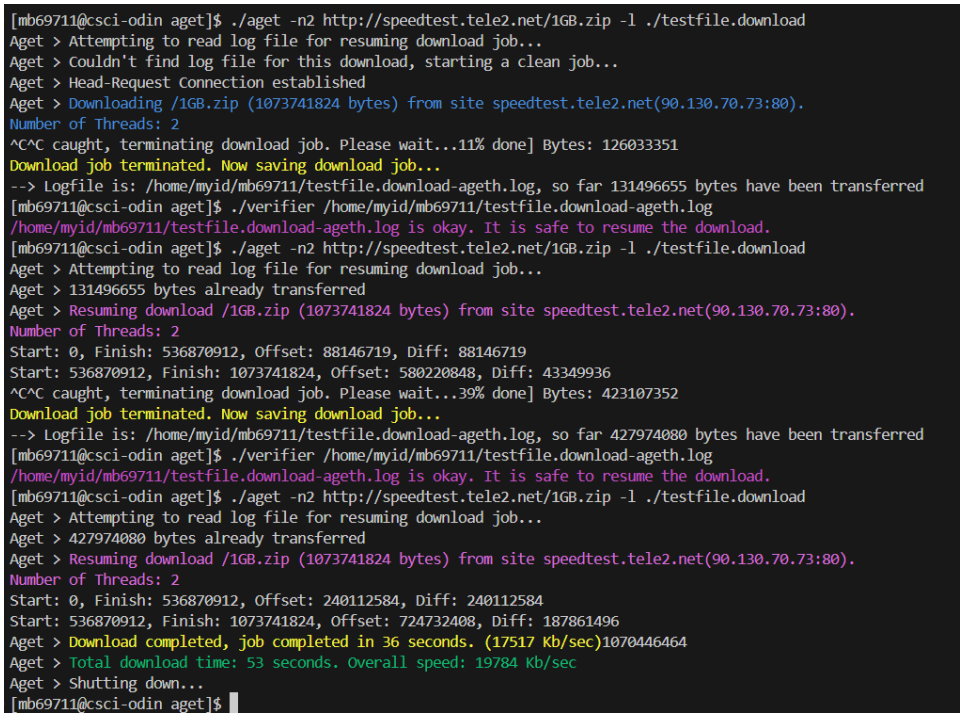
```
[mb69711@csci-odin aget]$ ./aget -n4 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > Couldn't find log file for this download, starting a clean job...
Aget > Head-Request Connection established
Aget > Downloading /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 4
File already exists! Overwrite?(y/n) y
^C^C caught, terminating download job. Please wait...23% done] Bytes: 123574572
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 220298548 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is okay. It is safe to resume the download.
[mb69711@csci-odin aget]$ ./aget -n4 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > 220298548 bytes already transferred
Aget > Resuming download /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 4
Start: 0, Finish: 134217728, Offset: 57999755, Diff: 57999755
Start: 134217728, Finish: 268435456, Offset: 188599939, Diff: 54382211
Start: 268435456, Finish: 402653184, Offset: 321884395, Diff: 53448939
Start: 402653184, Finish: 536870912, Offset: 457120827, Diff: 54467643
Aget > Download completed, job completed in 4 seconds. (77288 Kb/sec) 535616582
Aget > Total download time: 7 seconds. Overall speed: 74898 Kb/sec
Aget > Shutting down...
[mb69711@csci-odin aget]$
```

Then, I tried 9 threads, which was the biggest test. I interrupted the download at 26% completion, and the log accurately captured the state. When I resumed, it started right where it left off. The download finished in 8 seconds, and there were no issues.

```
[mb69711@csci-odin aget]$ ./aget -n9 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > Couldn't find log file for this download, starting a clean job...
Aget > Head-Request Connection established
Aget > Downloading /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 9
File already exists! Overwrite?(y/n) y
^C^C caught, terminating download job. Please wait...26% done] Bytes: 140128733
Download job terminated. Now saving download job...
--> Logfile is: /home/myid/mb69711/test.download-ageth.log, so far 250123157 bytes have been transferred
[mb69711@csci-odin aget]$ ./verifier /home/myid/mb69711/test.download-ageth.log
/home/myid/mb69711/test.download-ageth.log is okay. It is safe to resume the download.
[mb69711@csci-odin aget]$ ./aget -n9 http://ipv4.download.thinkbroadband.com/512MB.zip -l ./test.download
Aget > Attempting to read log file for resuming download job...
Aget > 250123157 bytes already transferred
Aget > Resuming download /512MB.zip (536870912 bytes) from site ipv4.download.thinkbroadband.com(80.249.99.148:80).
Number of Threads: 9
Start: 0, Finish: 59652323, Offset: 32455155, Diff: 32455155
Start: 59652323, Finish: 119304646, Offset: 81621055, Diff: 21968732
Start: 119304646, Finish: 178956969, Offset: 152095729, Diff: 32791083
Start: 178956969, Finish: 238609292, Offset: 203246844, Diff: 24289875
Start: 238609292, Finish: 298261615, Offset: 271093399, Diff: 32484107
Start: 298261615, Finish: 357913938, Offset: 323885098, Diff: 25623483
Start: 357913938, Finish: 417566261, Offset: 380253357, Diff: 22339419
Start: 417566261, Finish: 477218584, Offset: 450444224, Diff: 32877963
Start: 477218584, Finish: 536870912, Offset: 502511924, Diff: 25293340
Aget > Download completed, job completed in 5 seconds. (56005 Kb/sec) 529010286
Aget > Total download time: 8 seconds. Overall speed: 65536 Kb/sec
Aget > Shutting down...
[mb69711@csci-odin aget]$
```

**Name: Mayesha Bintha Mizan**
**ID-811281302**

Lastly, I tried with another video file. This file was successfully downloaded without any issues.




In all scenarios tested, including those involving large files, multiple threads, and repeated interruptions, the download resumption worked perfectly. No bugs were found during these tests, and the log files were accurately saved and verified each time.