Assignment 4

Kuruv Patel, U23AI051

February 4, 2025

1 Program 1

Observing Variable Addresses Across Multiple Threads

1.1 Introduction

This program demonstrates memory allocation and variable scope in a multi-threaded environment using POSIX threads (pthreads) in C. It explores how different types of variables—global, stack, heap, and thread-local—behave when accessed by multiple threads.

1.2 Source Code

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

// Thread-local variable
__thread int thread_local_var = 42;

// Global variable
int global_var = 100;

void *thread_function(void *arg)
{
    int thread_id = *(int *)arg;
    int stack_var = 10;                  // Stack variable
    int *heap_var = malloc(sizeof(int)); // Heap variable
    *heap_var = 20;

    printf("\nThread %d:\n", thread_id);
    printf("Stack variable address: %p (Value: %d)\n",
           (void *)&stack_var, stack_var);
    printf("Heap variable address: %p (Value: %d)\n",
           (void *)heap_var, *heap_var);
    printf("Thread-local variable address: %p (Value: %d)\n",
           (void *)&thread_local_var, thread_local_var);
    printf("Global variable address: %p (Value: %d)\n",
           (void *)&global_var, global_var);

    // Modify the thread-local variable to show it's unique per thread
    thread_local_var += thread_id;
    free(heap_var); // Clean up heap memory

    return NULL;
}

int main()
{
    pthread_t threads[3];
    int thread_ids[3] = {1, 2, 3};
```

```c

    printf("Main thread:\n");
    printf("Global variable address in main: %p\n",
           (void *)&global_var);
    printf("Thread-local variable address in main: %p\n",
           (void *)&thread_local_var);

    // Create three threads
    for (int i = 0; i < 3; i++)
    {
        if (pthread_create(&threads[i], NULL, thread_function,
                           &thread_ids[i]) != 0)
        {
            perror("Thread creation failed");
            return 1;
        }
    }

    // Wait for all threads to complete
    for (int i = 0; i < 3; i++)
    {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

1.3 Implementation

Thread Local Implementation using `__thread` keyword which creates a thread-local variable, we modify this local variable so that we can see that it is unique for every thread

Global Using a global variable accessible by all threads

Stack Using a stack-allocated variable

Heap Using dynamically allocated memory on the heap

PThread We use pthread to create threads

1.4 Experimental Results

```
kuruvpatel@Kuruv-Laptop:/mnt/c/Users/KURUV PATEL/OneDrive/Documents/LAB/OS Lab/Assignment 5$ gcc ./P1.c -o ./P1.out -pthread
kuruvpatel@Kuruv-Laptop:/mnt/c/Users/KURUV PATEL/OneDrive/Documents/LAB/OS Lab/Assignment 5$ ./P1.out
Main thread:
Global variable address in main: 0x559c06642010
Thread-local variable address in main: 0x7f8ccdf7a73c

Thread 1:
Stack variable address: 0x7f8ccdf78ea8 (Value: 10)
Heap variable address: 0x7f8cc8000b70 (Value: 20)
Thread-local variable address: 0x7f8ccdf796bc (Value: 42)
Global variable address: 0x559c06642010 (Value: 100)

Thread 2:
Stack variable address: 0x7f8ccd777ea8 (Value: 10)
Heap variable address: 0x7f8cc0000b70 (Value: 20)
Thread-local variable address: 0x7f8ccd7786bc (Value: 42)
Global variable address: 0x559c06642010 (Value: 100)

Thread 3:
Stack variable address: 0x7f8cccf76ea8 (Value: 10)
Heap variable address: 0x7f8cb8000b70 (Value: 20)
Thread-local variable address: 0x7f8cccf776bc (Value: 42)
Global variable address: 0x559c06642010 (Value: 100)
```

Figure 1: Output of the C code

```
kuruvpatel@Kuruv-Laptop:/mnt/c/Users/KURUV PATEL/OneDrive/Documents/LAB/OS Lab/Assignment 5$ strace -f -e clone ./P1.out
Main thread:
Global variable address in main: 0x5587d0e30010
Thread-local variable address in main: 0x7fb0c196f73c
strace: Process 588 attached
strace: Process 589 attached

Thread 1:

Thread 1:Stack variable address: 0x7fb0c196dea8 (Value: 10)
strace: Process 590 attached

Thread 1:Heap variable address: 0x7fb0bc000b70 (Value: 20)

Thread 1:Thread-local variable address: 0x7fb0c196e6bc (Value: 42)

Thread 2:

Thread 1:Global variable address: 0x5587d0e30010 (Value: 100)

Thread 3:

Thread 3:Stack variable address: 0x7fb0c096bea8 (Value: 10)

Thread 3:Heap variable address: 0x7fb0b8000b70 (Value: 20)
[pid   588] +++ exited with 0 +++

Thread 3:Thread-local variable address: 0x7fb0c096c6bc (Value: 42)

Thread 3:Global variable address: 0x5587d0e30010 (Value: 100)

Thread 2:Stack variable address: 0x7fb0c116cea8 (Value: 10)
[pid   590] +++ exited with 0 +++

Thread 2:Heap variable address: 0x7fb0b4000b70 (Value: 20)

Thread 2:Thread-local variable address: 0x7fb0c116d6bc (Value: 42)

Thread 2:Global variable address: 0x5587d0e30010 (Value: 100)
[pid   589] +++ exited with 0 +++
+++ exited with 0 +++
```

Figure 2: Output of the C code with strace

```
Main thread:
Global variable address in main: 0x555555558010
Thread-local variable address in main: 0x7ffff7da273c
[New Thread 0x7ffff7da16c0 (LWP 924)]

Thread 1:

Thread 1:Stack variable address: 0x7ffff7da0ea8 (Value: 10)

Thread 1:Heap variable address: 0x7ffff0000b70 (Value: 20)

Thread 1:Thread-local variable address: 0x7ffff7da16bc (Value: 42)

Thread 1:Global variable address: 0x555555558010 (Value: 100)
[New Thread 0x7ffff75a06c0 (LWP 925)]

Thread 2:

Thread 2:Stack variable address: 0x7ffff759fea8 (Value: 10)

Thread 2:Heap variable address: 0x7fffe8000b70 (Value: 20)

Thread 2:Thread-local variable address: 0x7ffff75a06bc (Value: 42)

Thread 2:Global variable address: 0x555555558010 (Value: 100)
[New Thread 0x7ffff6d9f6c0 (LWP 926)]

Thread 3:

Thread 3:Stack variable address: 0x7ffff6d9eea8 (Value: 10)

Thread 3:Heap variable address: 0x7fffec000b70 (Value: 20)

Thread 3:Thread-local variable address: 0x7ffff6d9f6bc (Value: 42)

Thread 3:Global variable address: 0x555555558010 (Value: 100)
[Thread 0x7ffff7da16c0 (LWP 924) exited]
[Thread 0x7ffff75a06c0 (LWP 925) exited]
[Thread 0x7ffff6d9f6c0 (LWP 926) exited]
[Inferior 1 (process 921) exited normally]
```

Figure 3: Output of the C code with gdb

1.5 Observation

- From the results we can observe that **Global Variable** *address* and *value* is constant in all the **threads**.
- The **Thread Local Variable** *address* is unique for all the threads, the *value* of the same is also constant for every thread even after modifying it in the code thus proving it is local to every thread.
- The **Stack and Heap Variable** *address* are unique for every threads.
- Using the **strace and gbd** commands we can observe when a thread is created and when it is killed.

1.6 Conclusion

1. **Stack Variables are Thread-Specific**
   Each thread has its own stack, so the stack variable (stack_var) inside thread_function has a different memory address for each thread. This means stack variables are not shared between threads.
2. **Heap Variables are Allocated Separately**
   Each thread dynamically allocates memory using malloc, and the heap variable (heap_var) has a unique memory address for each thread. This confirms that heap memory is shared among threads, but each allocation is independent.
3. **Thread-Local Variables are Unique per Thread**
   he __thread storage specifier ensures that thread_local_var is not shared between threads. Each thread gets its own instance of this variable, with a unique memory address, proving that thread-local storage (TLS) creates per-thread instances.
4. **Global Variables are Shared**
   The global variable global_var has the same memory address across all threads, confirming that global variables are shared between threads.
5. **Thread Execution Order is Non-Deterministic**
   Since the threads execute independently, their output order is not guaranteed. The exact order in which threads print their memory addresses and values may vary in different runs.

2 Program 2

Observing System Resource Utilization Using top Command

2.1 Introduction

This C program demonstrates significant CPU and memory usage. It dynamically allocates 3 GB of memory and performs a CPU-intensive computation. Running this program while monitoring system resources with 'top' will show increased memory and CPU consumption.

2.2 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

#define GB (1024L * 1024 * 1024)
#define ALLOC_SIZE (3 * GB)
#define ITERATIONS 100000000

int main()
{
    printf("Starting resource-intensive program...\n");

    // Allocate large memory block
    void *memory_block = malloc(ALLOC_SIZE);
    if (!memory_block)
    {
        perror("Memory allocation failed");
        return 1;
    }
    printf("Allocated %d MB of memory.\n", ALLOC_SIZE / (1024 * 1024));

    // Touch memory to ensure allocation is committed
    for (size_t i = 0; i < ALLOC_SIZE; i += 4096)
    {
        ((char *)memory_block)[i] = 'X';
    }
    printf("Memory initialized.\n");

    // CPU-intensive operation
    double result = 0.0;
    for (int i = 0; i < ITERATIONS; i++)
    {
        result += sqrt(i) * sin(i) * cos(i);
    }
    printf("Computation completed: %f\n", result);

    // Keep the program alive for observation in top
    printf("Sleeping for 10 seconds...\n");
    sleep(10);

    // Free allocated memory
    free(memory_block);
    printf("Memory freed. Exiting...\n");

    return 0;
}
```

2.3 Implementation

1. Allocates a large memory block and initializes it.

2. Performs a computationally expensive operation.

3. Sleeps for 10 seconds to allow observation in 'top'.

4. Frees the allocated memory and exits.

2.4 Experimental Results

```
top - 14:23:29 up  2:04,  1 user,  load average: 0.08, 0.03, 0.00
Tasks:  34 total,   1 running,  33 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.0 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   6785.2 total,   6017.3 free,    689.7 used,    307.7 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   6095.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    365 kuruvpa+  20   0    6720   5956   3604 S   0.3   0.1   0:00.28 bash
      1 root      20   0   21772  12940   9628 S   0.0   0.2   0:00.98 systemd
      2 root      20   0    2616   1444   1320 S   0.0   0.0   0:00.01 init-systemd(Ub
      6 root      20   0    2616    132    132 S   0.0   0.0   0:00.00 init
     51 root      19  -1   66824  17220  16096 S   0.0   0.2   0:00.76 systemd-journal
     93 root      20   0   24648   6568   4732 S   0.0   0.1   0:00.98 systemd-udevd
    106 root      20   0  153068    668    404 S   0.0   0.0   0:00.00 snapfuse
    107 root      20   0  152936    188     20 S   0.0   0.0   0:00.00 snapfuse
    108 root      20   0  153200   2240     44 S   0.0   0.0   0:00.00 snapfuse
    119 root      20   0  153068   2252     52 S   0.0   0.0   0:00.00 snapfuse
    124 root      20   0  153068   2252     36 S   0.0   0.0   0:00.00 snapfuse
    128 root      20   0  526756  14136    444 S   0.0   0.2   0:02.79 snapfuse
    150 systemd+  20   0   21452  11964   9772 S   0.0   0.2   0:00.13 systemd-resolve
    151 systemd+  20   0   91020   6468   5616 S   0.0   0.1   0:00.30 systemd-timesyn
    216 root      20   0    4236   2708   2476 S   0.0   0.0   0:00.03 cron
    217 message+  20   0    9632   5236   4536 S   0.0   0.1   0:00.30 dbus-daemon
    223 root      20   0 1986564  43524  19032 S   0.0   0.6   0:02.26 snapd
    224 root      20   0   17976   8440   7416 S   0.0   0.1   0:00.14 systemd-logind
    227 root      20   0 1755840  17992   9448 S   0.0   0.3   0:00.71 wsl-pro-service
    232 root      20   0    3160   1100   1012 S   0.0   0.0   0:00.01 agetty
    247 root      20   0    3116   1236   1144 S   0.0   0.0   0:00.00 agetty
    250 syslog    20   0  222508   7044   4468 S   0.0   0.1   0:00.25 rsyslogd
    253 root      20   0  107012  22312  12972 S   0.0   0.3   0:00.09 unattended-upgr
    358 root      20   0    2624    124      0 S   0.0   0.0   0:00.00 SessionLeader
    361 root      20   0    2624    132      0 S   0.0   0.0   0:00.17 Relay(365)
    371 root      20   0    6688   4684   3904 S   0.0   0.1   0:00.01 login
    435 kuruvpa+  20   0   20328  11564   9440 S   0.0   0.2   0:00.21 systemd
    436 kuruvpa+  20   0   21148   1728      0 S   0.0   0.0   0:00.00 (sd-pam)
    449 kuruvpa+  20   0    6072   5144   3488 S   0.0   0.1   0:00.00 bash
    849 polkitd   20   0  308160   9176   6496 S   0.0   0.1   0:00.07 polkitd
   1054 root      20   0    2624    128      0 S   0.0   0.0   0:00.00 SessionLeader
   1055 root      20   0    2624    136      0 S   0.0   0.0   0:00.03 Relay(1060)
   1060 kuruvpa+  20   0    6072   5332   3608 S   0.0   0.1   0:00.12 bash
   1081 kuruvpa+  20   0    9296   5168   3024 R   0.0   0.1   0:00.30 top
```

Figure 4: Before running the compiled code.

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1217 kuruvpa+  20   0 3149348   3.0g   1816 R 100.0  44.8   0:03.25 P2.out
      1 root      20   0   21772  12940   9628 S   0.0   0.2   0:01.01 systemd
```

Figure 5: Memory and CPU usage while the compiled code is running

2.5 Observation & Justifications

- After running the the compiled code which does CPU intensive work and allocates a large amount of memory, the CPU usage and MEM usage a significantly increased.
- Free memory available in the system around 3000MB after allocating 3000MB to the program.
- The process with PID 1217 which is the compiled code running is using the most CPU i.e. 100% and the most MEM i.e 45%.

2.6 Conclusion

1. Running **top** in a separate terminal provided real-time CPU, memory, and process statistics. Key metrics observed:
    (a) Total and free memory
    (b) CPU usage per process
    (c) Process priority and resource consumption
2. Impact of compiling and running a resource intensive program.

3 Program 3

Exploring strace for System Call Tracing in Linux

3.1 Introduction

Using **strace** command we will observer if commands are invoking system calls or not in the output.

3.2 Implementation

This program has been implemented by using the **strace** command and logging the output of the same for various commands that may or may not invoke system calls and to observe their behavior behind the scenes.

3.3 Results & Observation

```
    strace -o pwd_outpu.log -f pwd
```

The above strace command traces pwds calls:

```
    560   execve("/usr/bin/pwd", ["pwd"], 0x7ffd552ef638 /* 28 vars */) = 0
    560   getcwd("/mnt/c/Users/KURUV PATEL/OneDrive/Documents/LAB/OS Lab/Assignment 5",
    4096) = 68
    560   fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
    560   write(1, "/mnt/c/Users/KURUV PATEL/OneDriv"..., 68) = 68
    560   close(1)                         = 0
    560   exit_group(0)                    = ?
    560   +++ exited with 0 +++
```

From the above output we can observe that the command **pwd** does a write() system call.

```
   strace -o eq_outpu.log -f test 1 -eq 1
```

The above is a test command which evaluates an expression.

```
    622   execve("/usr/bin/test", ["test", "1", "-eq", "1"], 0x7fff0d8f5160 /* 28 vars */)
   = 0
    622   close(1)                         = 0
    622   close(2)                         = 0
    622   exit_group(0)                    = ?
    622   +++ exited with 0 +++
```

From the above output we can observe that the command **test** does not invoke a system call.

```
     strace -o cat_outpu.log -f head P2.c
```

The head command shows a files starting few lines

```
    702   execve("/usr/bin/head", ["head", "P2.c"], 0x7fff9d37b850 /* 28 vars */) = 0
    702   openat(AT_FDCWD, "P2.c", O_RDONLY) = 3
    702   read(3, "#include <stdio.h>\r\n#include <st"..., 8192) = 1166
    702   lseek(3, -976, SEEK_CUR)         = 190
    702   fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
    702   write(1, "#include <stdio.h>\r\n", 20) = 20
    702   write(1, "#include <stdlib.h>\r\n", 21) = 21
    702   write(1, "#include <unistd.h>\r\n", 21) = 21
    702   write(1, "#include <math.h>\r\n", 19) = 19
    702   write(1, "\r\n", 2)              = 2
    702   write(1, "#define GB (1024L * 1024 * 1024)"..., 34) = 34
```

```
    702   write(1, "#define ALLOC_SIZE (3 * GB)\r\n", 29) = 29
    702   write(1, "#define ITERATIONS 100000000\r\n", 30) = 30
    702   write(1, "\r\n", 2)                   = 2
    702   write(1, "int main()\r\n", 12)     = 12
    702   close(3)                            = 0
    702   close(1)                            = 0
    702   close(2)                            = 0
    702   exit_group(0)                       = ?
    702   +++ exited with 0 +++
```

From the above output we can observe that this command invokes openat(), read(), lseek(), write() system calls.

3.4 Conclusion

- Commands that manipulate files or interact with the environment tend to make more system calls, whereas simpler logical operations may not require them.
- The pwd command makes use of the write() system call to display the current working directory.
- test primarily operates at the shell level, evaluating expressions without requiring direct interaction with the kernel.
- head executes a new process, opens the specified file, reads its content, seeks within the file if necessary, and writes the output to standard output. This command invokes multiple system calls, including execve(), openat(), read(), lseek(), and write().

4 Program 4

Debugging a C Program with Loops, File I/O, and Memory Tracing

4.1 Introduction

This C program demonstrates core programming concepts including loops, file I/O with buffering, functions for basic operations, and pointer manipulation. We'll compile it with GCC's debugging flags and use GDB to trace execution, examine memory usage, and verify program behavior. The code showcases practical implementations of memory management, structured programming, and file handling while serving as a platform for learning debugging techniques.

4.2 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function prototypes
int findLargest(int a, int b);
void swapValues(int *a, int *b);
void writeToFile(const char *filename, int *numbers, int size);
void readFromFile(const char *filename);

int main() {
    // Initialize variables for demonstration
    int num1 = 25, num2 = 40;
    int numbers[] = {1, 2, 3, 4, 5};
    int *ptr = numbers;

    // Demonstrate loops
    printf("Using for loop to print array elements:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    // While loop with pointer arithmetic
    printf("\nUsing while loop with pointer arithmetic:\n");
    int count = 0;
    while (count < 5) {
        printf("%d ", *(ptr + count));
        count++;
    }
    printf("\n");

    // Find largest number
    printf("\nLargest between %d and %d is: %d\n", num1, num2, findLargest(num1, num2));

    // Swap values
    printf("\nBefore swap: num1 = %d, num2 = %d\n", num1, num2);
    swapValues(&num1, &num2);
    printf("After swap: num1 = %d, num2 = %d\n", num1, num2);

    // File I/O operations
    writeToFile("numbers.txt", numbers, 5);
    printf("\nReading from file:\n");
    readFromFile("numbers.txt");

    return 0;
}

// Function to find largest of two numbers
int findLargest(int a, int b) {
```

```c
    return (a > b) ? a : b;
}

// Function to swap two values using pointers
void swapValues(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to write numbers to file with buffering
void writeToFile(const char *filename, int *numbers, int size) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file for writing!\n");
        return;
    }

    // Set buffer for file operations
    char buffer[1024];
    setvbuf(file, buffer, _IOFBF, sizeof(buffer));

    for (int i = 0; i < size; i++) {
        fprintf(file, "%d\n", numbers[i]);
    }

    fclose(file);
}

// Function to read and display file contents
void readFromFile(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file for reading!\n");
        return;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }

    fclose(file);
}
```

4.3 Implementation

- Program finds the larges of two numbers, swaps two numbers, performs loops and writes a buffer to a file.
- We use gdb to perform debugging and set break points to observer memory changes.

4.4 Experimental Results

```
(gdb) break main
Breakpoint 1 at 0x1255: file ./P4.c, line 12.
(gdb) break swapValues
Breakpoint 2 at 0x1437: file ./P4.c, line 61.
(gdb) break writeToFile
Breakpoint 3 at 0x147b: file ./P4.c, line 68.
(gdb) run
Starting program: /mnt/c/Users/KURUV PATEL/OneDrive/Documents/LAB/OS Lab/Assignment 5/P4.out
```

Figure 6: GDB

```
Breakpoint 1, main () at ./P4.c:12
12      {
(gdb) next
14          int num1 = 25, num2 = 40;
(gdb) next
15          int numbers[] = {1, 2, 3, 4, 5};
(gdb) next
16          int *ptr = numbers;
(gdb) print num1
$1 = 25
(gdb) print num2
$2 = 40
(gdb) print numbers
$3 = {1, 2, 3, 4, 5}
(gdb) print &num1
$4 = (int *) 0x7fffffffd8b8
(gdb) next
19          printf("Using for loop to print array elements:\n");
(gdb) print ptr
$5 = (int *) 0x7fffffffd8d0
(gdb) print *ptr
$6 = 1
```

Figure 7: GDB

```
(gdb) continue
Continuing.
1 2 3 4 5

Using while loop with pointer arithmetic:
1 2 3 4 5

Largest between 25 and 40 is: 40

Before swap: num1 = 25, num2 = 40

Breakpoint 2, swapValues (a=0x7fffffffd8b8, b=0x7fffffffd8bc) at ./P4.c:61
61            int temp = *a;
(gdb) print a
$7 = (int *) 0x7fffffffd8b8
(gdb) print *a
$8 = 25
(gdb) contine
```

Figure 8: GDB

```
(gdb) continue
Continuing.
After swap: num1 = 40, num2 = 25

Breakpoint 3, writeToFile (filename=0x5555555560da "numbers.txt", numbers=0x7fffffffd8d0, size=5) at ./P4.c:68
68      {
(gdb) next
69          FILE *file = fopen(filename, "w");
(gdb) next
70          if (file == NULL)
(gdb) watch file
Hardware watchpoint 4: file
(gdb) next
78          setvbuf(file, buffer, _IOFBF, sizeof(buffer));
(gdb) next
80          for (int i = 0; i < size; i++)
(gdb) next
82              fprintf(file, "%d\n", numbers[i]);
(gdb) next
80          for (int i = 0; i < size; i++)
(gdb) continue
Continuing.

Watchpoint 4 deleted because the program has left the block in
which its expression is valid.
main () at ./P4.c:46
46          printf("\nReading from file:\n");
(gdb) continue
Continuing.

Reading from file:
1
2
3
4
5
[Inferior 1 (process 2009) exited normally]
```

Figure 9: GDB

```
(gdb) info locals
num1 = 25
num2 = 40
numbers = {1, 2, 3, 4, 5}
ptr = 0x7fffffffd8d0
count = 0
(gdb) next
Using for loop to print array elements:
20          for (int i = 0; i < 5; i++)
(gdb) info locals
i = 0
num1 = 25
num2 = 40
numbers = {1, 2, 3, 4, 5}
ptr = 0x7fffffffd8d0
count = 0
(gdb) where
#0  main () at ./P4.c:20
(gdb) next
22              printf("%d ", numbers[i]);
(gdb) where
#0  main () at ./P4.c:22
(gdb) next
20          for (int i = 0; i < 5; i++)
(gdb) continue
Continuing.
1 2 3 4 5

Using while loop with pointer arithmetic:
1 2 3 4 5

Largest between 25 and 40 is: 40

Before swap: num1 = 25, num2 = 40
After swap: num1 = 40, num2 = 25

Breakpoint 2, writeToFile (filename=0x5555555560da "numbers.txt", numbers=0x7fffffffd8d0, size=5) at ./P4.c:68
68      {
(gdb) where
#0  writeToFile (filename=0x5555555560da "numbers.txt", numbers=0x7fffffffd8d0, size=5) at ./P4.c:68
#1  0x00005555555553d3 in main () at ./P4.c:45
```

Figure 10: GDB-backtrace,info,locals

4.5 Observation

Memory Layout and Pointer Behavior: The variables num1 and num2 are stored in stack memory (visible from their addresses starting with 0x7ffffff...). The pointer arithmetic in the while loop (ptr + count) moves in increments of 4 bytes (sizeof(int)).

Backtrace: The nested function calls are visible in the backtrace. We can observe the exact sequence of loop iterations. Function parameters are passed correctly by value/reference

Memory Management: No dynamic memory allocation (malloc/free) is used. Stack-based variables are automatically managed.

File management Buffer for file I/O is statically allocated Buffer for file I/O is not immediately written to disk but held in the buffer. You can observe this by breaking after fprintf() calls The buffer flushes when it is full or when fclose() is called Program Flow.

4.6 Conclusion

GDB helps to

- Track variable changes in real-time
- Verify memory operations are safe
- Confirm control flow is correct
- Catch potential issues early
- Backtrack to function
- Provide with info for variables
- Provide memory addresses