

ECSE 427/COMP310 Lab7

Pthreads I

Jason Zixu Zhou

Feb-21-2025

What is a Thread?

- A thread is a lightweight process that shares resources like memory with other threads in the same process.
- **Threads vs. Processes:** Threads share the same memory space, processes have separate memory spaces.
- **Benefit of Multi-threading:** Concurrency and efficiency: multiple tasks can be handled simultaneously.

Thread Creation

- **Thread Creation with pthread_create:** pthread_create(&thread, NULL, function, arg);
- Creates a new thread that starts executing the given function.
- In the example, one thread handles even-indexed elements and the other handles odd-indexed elements.

Mutex Locks

- **What is a Mutex?**

- Ensures that only one thread can access a resource at a time.

- **Why use a mutex?**

- Prevent race conditions where multiple threads try to modify shared data simultaneously.

- **Functions:**

- `pthread_mutex_lock(&mutex);`
 - `pthread_mutex_unlock(&mutex);`

Condition Variables

- **Condition Variables:**

- `pthread_cond_wait(&cond, &mutex);`
- `pthread_cond_signal(&cond);`

- **Why are they used?**

- To coordinate the execution of threads.
- Ensures that threads alternate correctly when accessing shared resources.
- Example: One thread waits while the other processes data, and vice versa.

Data Sharing and Thread Safety

- **Shared Data in Threads:**

- The array and sums are shared between threads.
- This sharing can lead to data corruption without proper synchronization.

- **Mutex and Condition Variables for Safety:**

- Ensures data consistency while allowing threads to share information safely.

Thread Termination and Synchronization

- **Thread Termination with `pthread_join`:**

- Ensures that the main program waits for all threads to finish before exiting.

- **Why Synchronize?**

- Prevents the main thread from finishing before the worker threads.
- Allows threads to properly clean up resources before terminating.

Example

- Consider a randomly initialized array consisting of positive integers.
- You need to collect all the even indexed elements on one thread and all the odd indexed elements on a separate thread.
- Make sure to go over the indexes in increasing order without skipping any.
- Lastly, gather sum of only odd integers in even indexed thread and gather sum of only even integers in the odd indexed thread.

An example:

Some random array like = 3 1 55 4 5 8 7

Thread 1 will have:

Even indexed elements of the array: 3, 55, 5, 7

collect sum of only odd integers from this - $3+55+5+7 = 70$ is the output

thread 2

Odd indexed elements of the array: 1, 4, 8

collect sum of only even integers from this - $4+8 = 12$ is the output

Worker Functions

- **Worker Functions:**
 - evenWorker: Handles even-indexed elements, sums odd values.
 - oddWorker: Handles odd-indexed elements, sums even values.
- Parameters: Threads receive arguments like shared data through a structure.

Output Explanation

- **Program Objective:**

- Even-index thread: Collects odd integers from even indices.
- Odd-index thread: Collects even integers from odd indices.

- **Observe Output:**

- Output sum from each thread: Observe how the sums differ based on thread-specific logic.

How to run the program?

- To compile: `gcc pthreads-abz.c -lpthread -o second-example`
- To run: run the executable - `./name_of_the_executable` (`./second-example` in this case)
- Why do we need to use `-lpthread`?
 - We want the linker to be able to find the symbols defined in the pthread library.

What's the problem?

```
struct tracker *output = malloc(sizeof(struct tracker));  
output->arr = malloc(sizeof(int) * SIZE);  
output->arr = arr;  
output->evenSum = 0;  
output->oddSum = 0;  
  
pthread_mutex_init(&lock, NULL);  
pthread_cond_init(&cond, NULL);  
pthread_t thread[2];
```

Memory Allocation Problem

- malloc allocates memory for output->arr
- This memory is immediately overwritten by assigning arr to output->arr
- Causes a **memory leak** because the allocated memory is lost
- Program doesn't crash, but it wastes memory

Stack Memory Explanation

- Why Doesn't Using Stack Memory Cause a Crash?
 - arr is a local array on the stack
 - Its memory remains valid for the lifetime of the main function
 - Threads can safely access it while main is running
 - The program will only crash if arr goes out of scope while threads are still running

Thread Synchronization with Condition Variables

- `pthread_cond_wait` and `pthread_cond_signal` are used to coordinate thread execution
- One thread waits while the other processes its elements
- Prevents both threads from accessing the same index at the same time
- Works well for simple alternating tasks

Memory Cleanup and Freeing Resources

- The output structure is dynamically allocated with malloc
- It's not freed at the end of the program, causing a **memory leak**
- Memory leaks don't cause immediate errors, but they reduce efficiency
- Solution: Free output at the end of the program

Code Optimization and Expansion

- **Optimizing Code:**

- Large SIZE: How to manage performance.
- Efficient task splitting: Dynamically assign tasks to multiple threads.

- **Future Improvements:**

- More complex multi-threading tasks.
- Load balancing between threads.

Creating a thread

// About pthread_create and its arguments

https://man7.org/linux/man-pages/man3/pthread_create.3.html

SYNOPSIS top

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void *),
                  void *restrict arg);
```

Compile and link with -pthread.

The attr argument points to a pthread_attr_t structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using pthread_attr_init(3) and related functions. If attr is NULL, then the thread is created with default attributes.

- Four arguments to pthread_create
- Pointer to the thread
- Attributes to describe life cycle of the thread
- Function which the thread should execute
- Arguments to the previously mentioned function

Join threads

- Join – waiting until thread is done with its execution
- A call to `pthread_join` blocks the calling thread until the thread with identifier equal to the first argument terminates.
- The first argument to `pthread_join()` is the identifier of the thread to join. The second argument is a void pointer.
- `pthread_join(pthread_t tid, void * return_value);`
- If the `return_value` pointer is non-NULL, `pthread_join` will place at the memory location pointed to by `return_value`, the value passed by the thread `tid` through the `pthread_exit` call.
- Since we don't care about return value of the thread, we set it to NULL.

Launching Threads and Routine Function for pthread_create

- **Launching Threads:**
- Use pthread_create to launch threads.
- Each thread runs a specific routine (function) defined by the programmer.
- Routine for even-indexed elements: evenWorker
- Routine for odd-indexed elements: oddWorker
 - pthread_create(&thread[0], NULL, evenWorker, output);
 - pthread_create(&thread[1], NULL, oddWorker, output);

Why Not Pass the Index Address Directly?

- **Issue with Passing Index Address:**

- Multiple threads might access the same memory address.
- This can lead to **data races** and unpredictable behavior.
- Example in the code: `int pos = 0;` is shared by both threads and protected with a mutex lock.
- Solution: Use dynamic memory or pass separate values for each thread.

Why Allocate Memory for Thread Data?

- **Allocating Memory for Thread Arguments:**
- Ensures each thread has independent data.
- Prevents sharing the same memory address by multiple threads.
- In the code: `struct tracker *output = malloc(sizeof(struct tracker));`
- Dynamic allocation guarantees each thread gets its own copy of the data.

Passing Array Elements by Address

- **Directly Pass Element Addresses:**
- Instead of passing entire array, pass individual element addresses.
- This gives each thread its own element to process.
- Example: `pthread_create(&thread[i], NULL, thread_function, &array[i]);`
- Each thread works on its own element without needing additional memory allocation.

Returning Values from Threads

- **Using pthread_exit to Return Values:**
- pthread_exit allows threads to return values.
- The main thread retrieves the result using pthread_join.
 - `int *result = malloc(sizeof(int));`
 - `*result = some_calculation();`
 - `pthread_exit(result);`
 - `pthread_join(thread, (void**)&result);`

When to Free Memory

- Memory should be freed after all threads have finished.
- Example in the code:
- `free(output);` is called after `pthread_join`.
- If memory is freed too early, threads might access invalid memory.
- Always free memory after using `pthread_join` to ensure threads are done.