Misato Seki

Exploring Multi-threading Concepts

Signals, Thread Cancellation, Thread-local Storage, and Scheduler Activations

Report
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Bachelor of Engineering, Information Technology
March 2023

ABSTRACT

Centria University of Applied Sciences	Date March 2023	Author Misato Seki
Degree programme Bachelor of Engineering, Information Technology		
Name of thesis Exploring Multi-threading Concepts		
Centria supervisor		**Pages** 7 + 1
Instructor representing commissioning institution or company		

In this report, we will learn Signals, Thread Cancellation, Thread-local Storage, and Scheduler Activations to deepen our understanding of Multi-threading Concepts.

As a conclusion, **signal** is a mechanism used to notify a process or thread that an event has occurred. When we ran the program we created, the receiving of a signal allowed the thread to call a pre-registered signal handler function. We found that signals were used to debug or abort the program or to trigger certain actions.

Thread cancelation is to suspend or terminate a running thread. When we ran the program we created, the thread was able to perform the cancel process by receiving of a cancel request. We found that thread cancelation is used to handle program errors, release resources, or control program behavior.

Thread local storage is an area of memory allocated separately for each thread. It is used to store state held by threads. When we ran the program we created, the data stored in TLS was not accessed by other threads. We found that TLS is used as an alternative to global variables or to maintain thread-specific state.

Scheduler activation is a mechanism used by the thread scheduler to determine which resources to allocate to a thread before it begins execution. Scheduler activation was first introduced in the Mach operating system and is used in some modern operating systems such as FreeBSD. However, we found that it has not support in other operating systems such as macOS, Linux, and Windows.

Key words

ABSTRACT
CONTENTS

1 INTRODUCTION

In this report, we will examine Signals, Thread Cancellation, Thread-local Storage, and Scheduler Activations to deepen our understanding of Multi-threading Concepts. We will also write, run, and observe actual code.

2 SIGNAL

In the multi-threading, signals are sent to the entire process and may be delivered to any thread. Therefore, when handling signals, signals must be handled in a thread-safe manner.
In general, when handling signals in a multithreaded program, a thread-safe synchronization object (e.g., **std**::**atomic** or **std**::**mutex**) is used in the signal handler function to tell all threads that a signal has occurred and to handle it appropriately.
It is also common to handle signals in a multi-threaded program by creating a dedicated thread to handle and wait for the signal. In this case, synchronization between threads is required so that the signal is processed by the dedicated thread when a signal occurs.

In the following program, **signal_handler** is registered to handle **SIGINT** signals.
When a signal is received, the signal handler function is called in the context of the currently executing thread. In the following program, if a **SIGINT** signal is received while **check_for_termination** is running, the signal handler is executed in the context of that thread.
However, by using the **terminate_program** which is **std**::**atomic**<**bool**> variable in the program, both threads have a consistent view of the program state, and when the signal handler sets **terminate_program** to true, both threads are terminated.

```cpp
#include <iostream>
#include <csignal>
#include <thread>
#include <atomic>
#include <chrono>
#include <unistd.h>

std::atomic<bool> terminate_program(false);

void signal_handler(int signal_num) {
    std::cout << "Received signal: " << signal_num << std::endl;
    terminate_program = true;
}

void check_for_termination() {
    while (!terminate_program) {
        // Do some work
        std::cout << "Thread 2: Waiting for a signal..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    std::cout << "Thread 2: Terminating gracefully..." << std::endl;
}
```

```cpp
int main() {
    std::cout << "My process ID is: " << getpid() << std::endl;

    // Register a signal handler for SIGINT (Ctrl+C)
    signal(SIGINT, signal_handler);

    // Create a thread to check for termination
    std::thread t2(check_for_termination);

    // Loop forever in the main thread
    while (!terminate_program) {
        std::cout << "Thread 1: Waiting for a signal..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    // Wait for the check_for_termination thread to finish
    t2.join();

    std::cout << "Thread 1: Terminating gracefully..." << std::endl;

    return 0;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
My process ID is: 89061
Thread 1: Waiting for a signal...
Thread 2: Waiting for a signal...
Thread 1: Waiting for a signal...
Thread 2: Waiting for a signal...
Thread 2: Waiting for a signal...
Thread 1: Waiting for a signal...
^CReceived signal: 2
Thread 2: Terminating gracefully...
Thread 1: Terminating gracefully...
```

PICTURE 1. C++ code using SIGNAL concept

3 THREAD CANCELATION

Asynchronous cancellation is a method by which a thread is immediately canceled, and the thread can be suspended when the cancellation is requested. In the following program, if a loop in the **worker_thread** function is running, the loop is interrupted and the **worker_thread** is terminated immediately.
Deferred cancellation is a method of waiting until a thread reaches a "cancellation point" before suspending the thread. The cancellation point is a mark set at a specific location in the program, and the thread is not cancelled before reaching the cancellation point. In the program below, if a loop in the **worker_thread** function is running, the **worker_thread** is not interrupted until the loop ends, and then the **worker_thread** exits.
In other words, if the async option is selected in this program, the thread is separated and immediately canceled; if the deferred option is selected, the thread is joined, waits for a cancellation point, and then interrupted.

```cpp
#include <iostream>
#include <csignal>
#include <thread>
#include <chrono>
#include <atomic>
#include <unistd.h>

std::atomic<bool> terminate_flag(false);

void signal_handler(int signal_num) {
    std::cout << "Received signal: " << signal_num << std::endl;
    terminate_flag = true;
}

void worker_thread() {
    while (!terminate_flag) {
        std::cout << "Working..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    std::cout << "Terminating worker thread gracefully." << std::endl;
}

int main(int argc, char* argv[]) {
    std::cout << "My process ID is: " << getpid() << std::endl;

    // Register a signal handler for SIGINT (Ctrl+C)
    signal(SIGINT, signal_handler);

    if (argc > 1 && std::string(argv[1]) == "async") {
        // Use asynchronous cancellation
        std::cout << "Using asynchronous cancellation." << std::endl;
        std::thread worker(worker_thread);
        worker.detach();
        while (!terminate_flag) {
            // Wait for user input to set the terminate flag
            std::string input;
            std::getline(std::cin, input);
            if (input == "q") {
                terminate_flag = true;
            }
        }
        std::cout << "Terminating main thread gracefully." << std::endl;
    } else if (argc > 1 && std::string(argv[1]) == "deferred") {
        // Use deferred cancellation
        std::cout << "Using deferred cancellation." << std::endl;
        std::thread worker(worker_thread);
        while (!terminate_flag) {
            // Wait for user input to set the terminate flag
            std::string input;
            std::getline(std::cin, input);
            if (input == "q") {
                terminate_flag = true;
                worker.join();
                std::cout << "Terminating main thread gracefully." << std::endl;
            }
        }
    } else {
        std::cerr << "Invalid command line argument. Must be 'async' or 'deferred'." << std::endl;
        return 1;
    }

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
misatoseki@MisatonoMacBook-Air Session12 % ./Cancellation.out async
My process ID is: 90749
Using asynchronous cancellation.
Working...
Working...
Working...
^CReceived signal: 2
Terminating worker thread gracefully.
q
Terminating main thread gracefully.
misatoseki@MisatonoMacBook-Air Session12 % ./Cancellation.out deferred
My process ID is: 90836
Using deferred cancellation.
Working...
Working...
Working...
Working...
^CReceived signal: 2
Terminating worker thread gracefully.
q
Terminating main thread gracefully.
```

PICTURE 2. C++ code using THREAD CANCELATION concept

4 THREAD CANCELLATION IN JAVA.

The thread cancellation mechanism in Java differs from that in C++. In C++, a checkpoint must be periodically set in the thread to explicitly cancel a thread. Java, on the other hand, provides the interrupt() method to achieve thread cancellation.

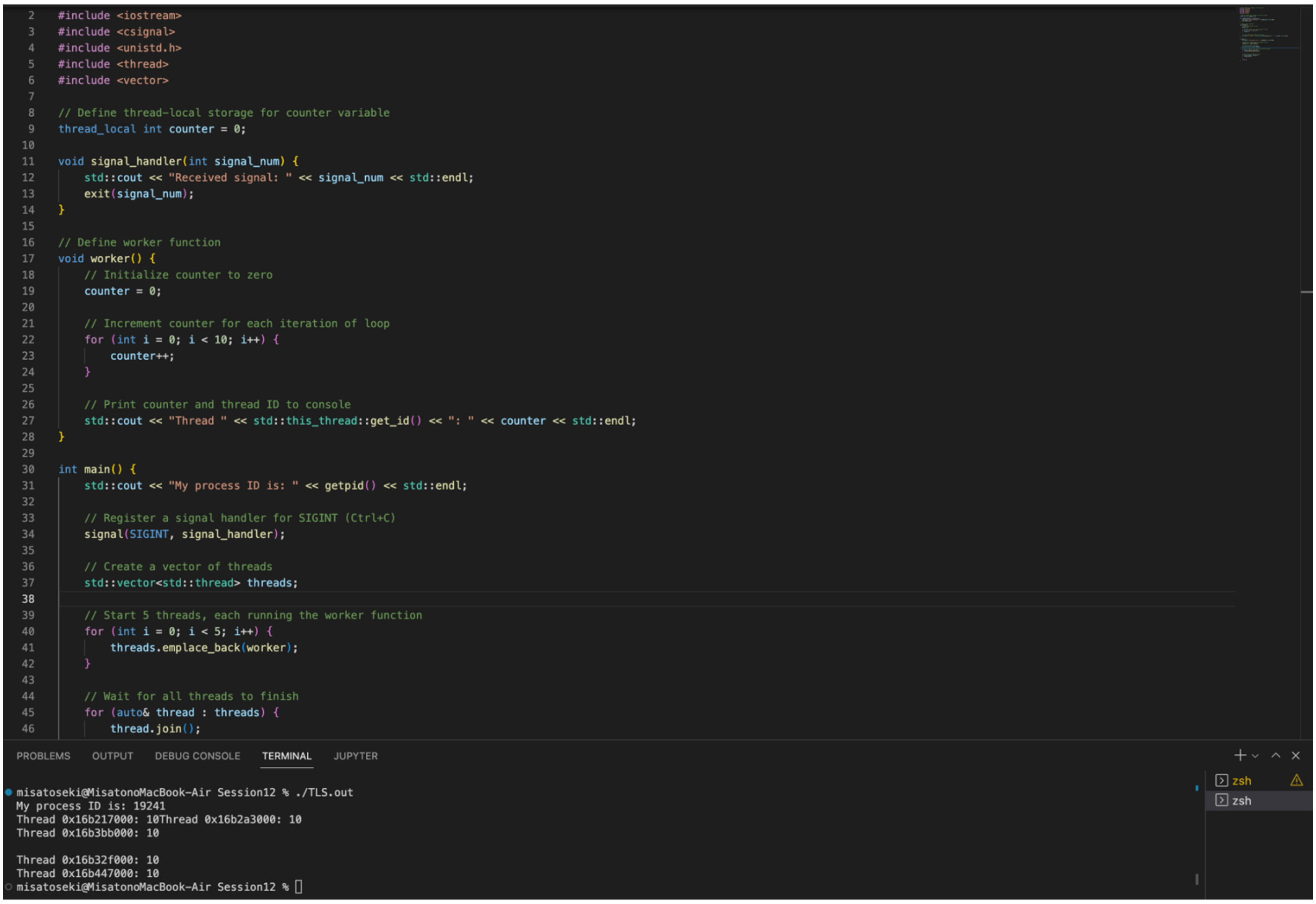
The interrupt() method generates an interrupt for a thread. This interrupt is checked while the thread is running to ensure that the thread is properly canceled. A thread can detect cancellation by being set to the interrupted state during its check.

If a thread is in the interrupted state, the thread may throw an InterruptedException instead of continuing normal execution. InterruptedException can be handled within the thread, and the cleanup process necessary to handle the thread cancellation can be performed.

5 THREAD-LOCAL STORAGE (TLS)

In the multithreaded programming, variables shared among threads have a problem. When multiple threads access and update the value of a variable at the same time, a race issue can occur. Such a race issue can produce inaccurate results or cause the program to stop working.
Thread-local storage (TLS) is a feature that allows each thread to store variables in its own separate memory area. In other words, there is no need to share variable values among threads by using TLS. As a result, race issues are avoided, and safe and accurate parallel processing is achieved.

In the following code, each thread stores the **counter** variable in its own thread-local storage. Each thread increments its own **counter** variable each time it executes a loop. Thus, by using TLS, each thread can safely manipulate its own counter and race issues are avoided.



```cpp
#include <iostream>
#include <csignal>
#include <unistd.h>
#include <thread>
#include <vector>

// Define thread-local storage for counter variable
thread_local int counter = 0;

void signal_handler(int signal_num) {
    std::cout << "Received signal: " << signal_num << std::endl;
    exit(signal_num);
}

// Define worker function
void worker() {
    // Initialize counter to zero
    counter = 0;

    // Increment counter for each iteration of loop
    for (int i = 0; i < 10; i++) {
        counter++;
    }

    // Print counter and thread ID to console
    std::cout << "Thread " << std::this_thread::get_id() << ": " << counter << std::endl;
}

int main() {
    std::cout << "My process ID is: " << getpid() << std::endl;

    // Register a signal handler for SIGINT (Ctrl+C)
    signal(SIGINT, signal_handler);

    // Create a vector of threads
    std::vector<std::thread> threads;

    // Start 5 threads, each running the worker function
    for (int i = 0; i < 5; i++) {
        threads.emplace_back(worker);
    }

    // Wait for all threads to finish
    for (auto& thread : threads) {
        thread.join();
```



```
misatoseki@MisatonoMacBook-Air Session12 % ./TLS.out
My process ID is: 19241
Thread 0x16b217000: 10Thread 0x16b2a3000: 10
Thread 0x16b3bb000: 10

Thread 0x16b32f000: 10
Thread 0x16b447000: 10
misatoseki@MisatonoMacBook-Air Session12 %
```

PICTURE 3. C++ code using TLS concept

6 SCHEDULER ACTIVATION

Scheduler activation is the mechanism used to schedule LWPs and differs from traditional thread scheduling. In traditional thread scheduling, the operating system schedules threads, which execute within the address space of the process. With scheduler activation, on the other hand, the application controls the scheduling of threads, and the threads execute within the LWP.
Specifically, the application uses Scheduler Activation to schedule threads in the LWP. Applications can set priorities for threads in the LWP and switch threads within the LWP. Applications can also control the number of threads running in the LWP.

In this code, a vector of **lwpid_t** values are created to represent each LWP. The **lwp_create**() function is used to create each LWP, passing in a pointer to the **worker**() function as the thread function. The newly created LWP is added to the vector and then resume it using the **lwp_continue**() function.
Within the **worker**() function, we use **thr_self**() to get the ID of the current LWP. Then we suspend the LWP using **lwp_suspend**().

```cpp
#include <iostream>
#include <csignal>
#include <unistd.h>
#include <thread>
#include <vector>
#include <thread.h>

// Define thread-local storage for counter variable
thread_local int counter = 0;

void signal_handler(int signal_num) {
    std::cout << "Received signal: " << signal_num << std::endl;
    exit(signal_num);
}

// Define worker function
void worker() {
    // Initialize counter to zero
    counter = 0;

    // Increment counter for each iteration of loop
    for (int i = 0; i < 10; i++) {
        counter++;
    }

    // Print counter and thread ID to console
    std::cout << "Thread " << thr_self() << ": " << counter << std::endl;

    // Suspend the LWP
    lwp_suspend();
}
```

```cpp
int main() {
    std::cout << "My process ID is: " << getpid() << std::endl;

    // Register a signal handler for SIGINT (Ctrl+C)
    signal(SIGINT, signal_handler);

    // Create a vector of LWPs
    std::vector<lwpid_t> lwps;

    // Start 5 LWPs, each running the worker function
    for (int i = 0; i < 5; i++) {
        // Create the LWP and add it to the vector
        lwpid_t lwp;
        if (lwp_create((thread_func_t) worker, 0, &lwp, NULL) == -1) {
            std::cerr << "Failed to create LWP" << std::endl;
            return 1;
        }
        lwps.push_back(lwp);

        // Resume the LWP
        lwp_continue(lwp);
    }

    // Wait for all LWPs to finish
    for (auto& lwp : lwps) {
        // Suspend the LWP
        lwp_suspend();

        // Destroy the LWP
        if (lwp_destroy(lwp) == -1) {
            std::cerr << "Failed to destroy LWP" << std::endl;
            return 1;
        }
    }

    return 0;
}
```

PICTURE 4. C++ code using SCHEDULER ACTIVATION concept

7 CONCLUSION

A **signal** is a mechanism used to notify a process or thread that an event has occurred. When we ran the program we created, the receiving of a signal allowed the thread to call a pre-registered signal handler function. We found that signals were used to debug or abort the program or to trigger certain actions.

Thread cancelation is to suspend or terminate a running thread. When we ran the program we created, the thread was able to perform the cancel process by receiving of a cancel request. We found that thread cancelation is used to handle program errors, release resources, or control program behavior.

Thread local storage is an area of memory allocated separately for each thread. It is used to store state held by threads. When we ran the program we created, the data stored in TLS was not accessed by other threads. We found that TLS is used as an alternative to global variables or to maintain thread-specific state.

Scheduler activation is a mechanism used by the thread scheduler to determine which resources to allocate to a thread before it begins execution. Scheduler activation was first introduced in the Mach operating system and is used in some modern operating systems such as FreeBSD. However, we found that it has not support in other operating systems such as macOS, Linux, and Windows.

REFERENCES

chatGPT
https://chat.openai.com/chat