Lab Report 3

Due November 5th, 2025

By

Tudor Cosmin Suciu

I certify that this
submission is my original work and meets
the Faculty's Expectations of Originality

ID: 40179863

November 5th, 2025

COEN 320

Real Time Systems

Section FL-X

Concordia University

INTRODUCTION

In this lab experiment we will see key concepts of real-time systems like synchronous messaging between processes, server-client model, inter-process synchronization of shared resources and semaphores The objective of this lab is to learn and get used to the functioning and utility of these concepts and to explore what would happen without them.

PART 1: Synchronous message passing (Client-Server Model)

Code:

Client.cpp:

```
// client.cpp
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>          // For sleep()
#include <sys/neutrino.h>    // For MsgSend, ConnectDetach
#include <sys/dispatch.h>    // For name_open, name_close
#include <iostream>
#include <ctime>
#include <chrono>
#include <iomanip>
#include <sstream>
#include <cstdio>

using namespace std;

#define MAX_MSG_SIZE 256
#define MSG_TYPE_HELLO 1
#define MSG_TYPE_ADD   2
#define MSG_TYPE_QUIT  99

// Message structures (must match server)
typedef struct {
    int  type;
    char text[MAX_MSG_SIZE];
    int  num1;
    int  num2;
} ClientMsg;

typedef struct {
    char response[MAX_MSG_SIZE];
    int  result;
} ServerReply;

//Get current timestamp
string current_timestamp() {
```

```
    using namespace chrono;

    auto now = system_clock::now();

    // Convert to local time (HH:MM:SS)
    time_t tt = system_clock::to_time_t(now);
    tm tm{};

    localtime_r(&tt, &tm); //for running on linux raspberry Pi

    // Extract milliseconds
    auto ms = duration_cast<milliseconds>(now.time_since_epoch()) % 1000;

    // Build string
    ostringstream oss;
    oss << put_time(&tm, "%H:%M:%S") << '.'
        << setfill('0') << setw(3) << ms.count();
    return oss.str();
}

int main(void) {
    int coid;           // Connection ID
    ClientMsg msg;
    ServerReply reply;

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Connecting to server named 'cosmin_server'...\n");

    // 1. Connect to the named server channel
    coid = name_open("cosmin_server", 0);
    if (coid == -1) {
        perror("Client: name_open failed (server may not be running)");
        return EXIT_FAILURE;
    }

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Connected to server (connection ID: %d)\n", coid);

    // --- Send a "HELLO" message ---
    msg.type = MSG_TYPE_HELLO;
    strcpy(msg.text, "Hello from client!");

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Sending HELLO message...\n");
    if (MsgSend(coid, &msg, sizeof(msg), &reply, sizeof(reply)) == -1) {
        perror("Client: MsgSend (HELLO) failed");
        name_close(coid);
        return EXIT_FAILURE;
    }

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Server replied: %s\n", reply.response);
    sleep(1);

    // --- Send an "ADD" message ---
    msg.type = MSG_TYPE_ADD;
```

```
    msg.num1 = 10;
    msg.num2 = 25;

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Sending ADD message (%d + %d)...\n", msg.num1, msg.num2);
    if (MsgSend(coid, &msg, sizeof(msg), &reply, sizeof(reply)) == -1) {
        perror("Client: MsgSend (ADD) failed");
        name_close(coid);
        return EXIT_FAILURE;
    }

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Server replied: %s (Result: %d)\n", reply.response, reply.result);
    sleep(1);

    // --- Send a "QUIT" message to the server ---
    msg.type = MSG_TYPE_QUIT;

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Sending QUIT message to server...\n");
    if (MsgSend(coid, &msg, sizeof(msg), &reply, sizeof(reply)) == -1) {
        perror("Client: MsgSend (QUIT) failed");
        name_close(coid);
        return EXIT_FAILURE;
    }

    printf("%s - ", current_timestamp().c_str());
    printf("Client: Server replied: %s\n", reply.response);

    // 2. Close the connection
    name_close(coid);
    delay(2000);
    printf("%s - ", current_timestamp().c_str());
    printf("Client: Disconnected. Exiting.\n");

    return EXIT_SUCCESS;
}
```

Server.cpp:

```
// server.cpp
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>   // For ChannelCreate, MsgReceive, MsgReply
#include <sys/dispatch.h>
#include <string.h>         // For strcmp, strcpy
#include <errno.h>          // For EOK (often defined as 0 or in specific sys headers)
#include <iostream>
#include <ctime>
#include <chrono>
#include <iomanip>
#include <sstream>
#include <cstdio>
using namespace std;

#define MAX_MSG_SIZE 256
#define MSG_TYPE_HELLO 1
#define MSG_TYPE_ADD   2
#define MSG_TYPE_QUIT  99

// Structure for messages
typedef struct {
    int  type;
    char text[MAX_MSG_SIZE];
    int  num1;  // For MSG_TYPE_ADD
    int  num2;  // For MSG_TYPE_ADD
} ClientMsg;

// Structure for replies
typedef struct {
    char response[MAX_MSG_SIZE];
    int  result;  // For MSG_TYPE_ADD
} ServerReply;

//Get current timestamp
string current_timestamp() {
    using namespace chrono;

    auto now = system_clock::now();

    // Convert to local time (HH:MM:SS)
    time_t tt = system_clock::to_time_t(now);
    tm tm{};

    localtime_r(&tt, &tm); //for running on linux raspberry Pi

    // Extract milliseconds
    auto ms = duration_cast<milliseconds>(now.time_since_epoch()) % 1000;

    // Build string
    ostringstream oss;
    oss << put_time(&tm, "%H:%M:%S") << '.'
        << setfill('0') << setw(3) << ms.count();
    return oss.str();
```

```
}

int main(void) {
    int chid;       // Channel ID (not explicitly used with name_attach)
    int rcvid;      // Receive ID
    ClientMsg msg;
    ServerReply reply;

    printf("%s - ", current_timestamp().c_str());
    printf("Server: Starting...\n");

    // 1. Create a channel and attach a name to it (QNX specific)
    name_attach_t *attach = name_attach(NULL, "cosmin_server", 0);
    if (attach == NULL) {
        perror("Server: name_attach failed");
        return EXIT_FAILURE;
    }

    printf("%s - ", current_timestamp().c_str());
    printf("Server: Channel created with ID %d. Waiting for clients...\n", attach->chid);

    while (1) {
        // 2. Wait for a message
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);
        if (rcvid == -1) {
            perror("Server: MsgReceive failed");
            break;
        }

        if (rcvid == 0) {  // Pulse received (e.g., from timer, not a message)
            printf("%s - ", current_timestamp().c_str());
            printf("Server: Received a pulse (code %d).\n", msg.type); // msg.type here is actually pulse.code
            continue;
        }

        // A message was received
        printf("%s - ", current_timestamp().c_str());
        printf("Server: Received message from rcvid %d. Type: %d\n", rcvid, msg.type);

        // Process the message
        memset(&reply, 0, sizeof(reply));  // Clear reply buffer

        switch (msg.type) {
            case MSG_TYPE_HELLO:
                printf("%s - ", current_timestamp().c_str());
                printf("Server: Client says: %s\n", msg.text);
                strcpy(reply.response, "Hello Client! Message received.");
                break;

            case MSG_TYPE_ADD:
                printf("%s - ", current_timestamp().c_str());
                printf("Server: Client requests addition of %d and %d.\n", msg.num1, msg.num2);
                reply.result = msg.num1 + msg.num2;
                sprintf(reply.response, "Addition result: %d", reply.result);
                break;
```

```
            case MSG_TYPE_QUIT:
                printf("%s - ", current_timestamp().c_str());
                printf("Server: Client requested quit. Exiting.\n");
                strcpy(reply.response, "Server shutting down.");
                MsgReply(rcvid, EOK, &reply, sizeof(reply));  // Reply before exit
                goto end_server_loop;  // Exit loop

            default:
                printf("%s - ", current_timestamp().c_str());
                printf("Server: Unknown message type %d\n", msg.type);
                strcpy(reply.response, "Unknown message type.");
                break;
        }

        // 3. Reply to the client
        // COMMENT THIS OUT TO SHOW THAT THE CLIENT BLOCKS IF SERVER CAN'T REPLY
        MsgReply(rcvid, EOK, &reply, sizeof(reply));  // EOK = success (0)
    }

end_server_loop:
    printf("%s - ", current_timestamp().c_str());
    printf("Server: Shutting down.\n");
    ChannelDestroy(attach->chid);   // Use attach->chid instead of uninitialized 'chid'
    name_detach(attach, 0);

    return EXIT_SUCCESS;
}
```

Execution Logs:

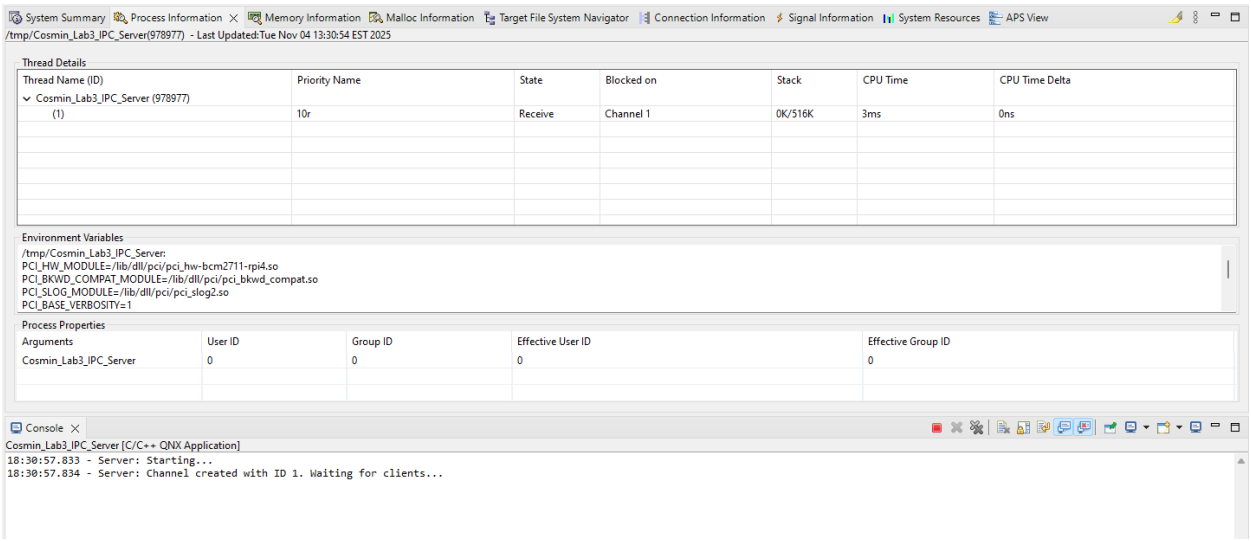


Figure 1: Execution log of Server starting and waiting on Client to start.

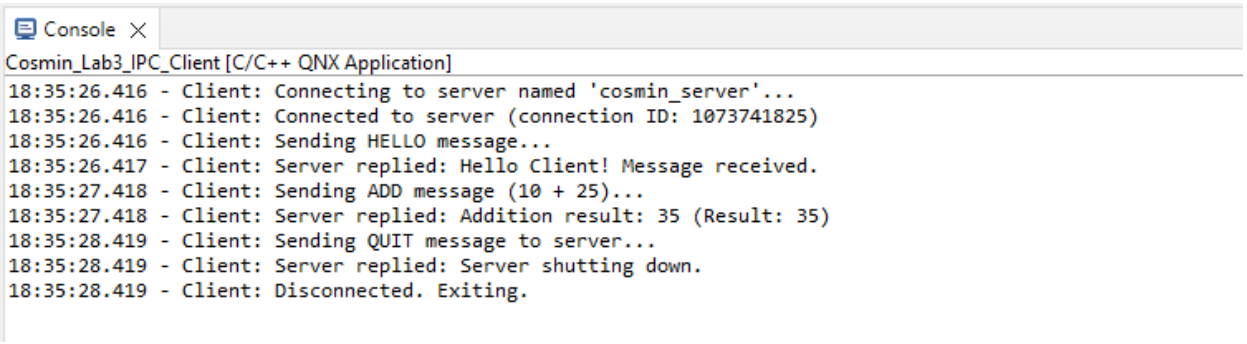


Figure 2: Execution log of Client running until the end

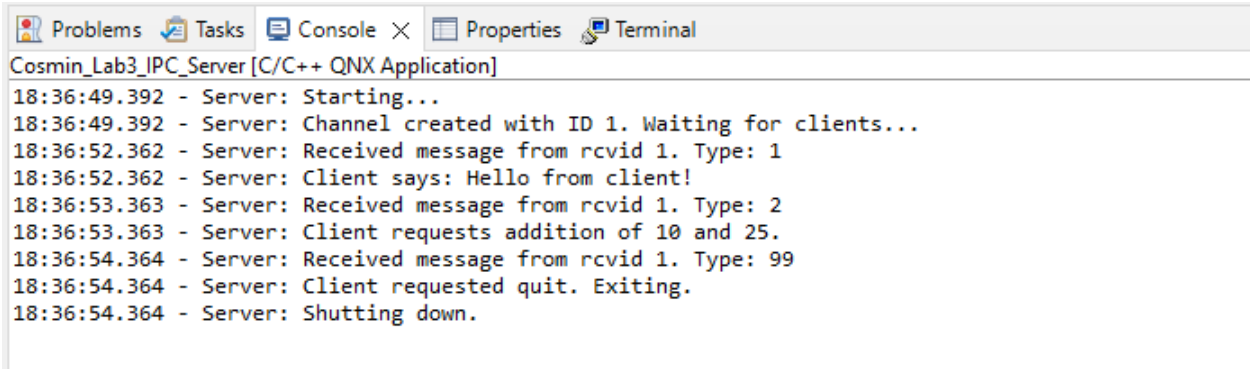


Figure 3: Execution log of Server running until the end

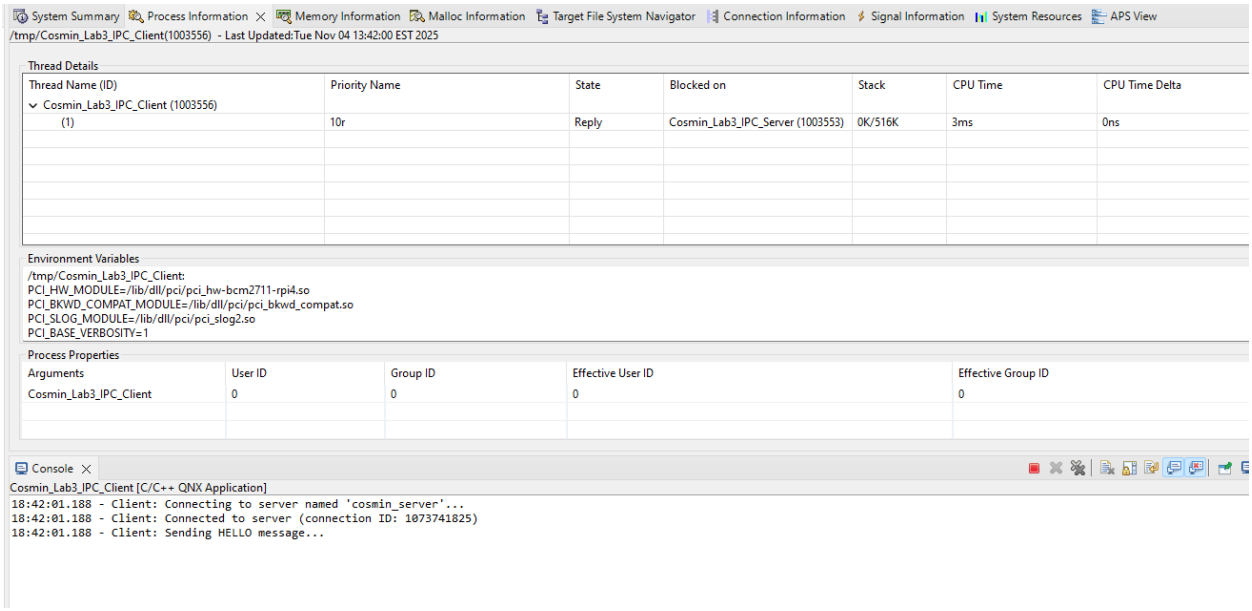


Figure 4: Execution log of Client blocking when Server doesn't reply

Figure 1 shows the Server starting and blocking to wait for Clients. Figure 2 and 3 shows the entire console execution of the Client and Server processes respectively. At first, the Client connects to the Server named "cosmin_server", produces a connection ID, sends a HELLO message and blocks to wait on the Server reply. The blocking behavior is clearly visible in Figure 4, where I intentionally took away the Server's ability to respond in order to demonstrate blocking. Next, the Server replies with an acknowledgement and the Client can move on to sending the next message. The two continue to exchange messages until the client sends the QUIT message which makes the Server shutdown.

Discussion:

In the QNX RTOS most inter-process communication is done through synchronous message-passing because it is a deterministic and safe way of communicating which avoids race conditions and reduces risks associated with standard shared memory.

The server creates a channel which acts like an address where clients can send messages to. In order to communicate through a channel, a client must first connect to it using ConnectAttach(), returning a connection ID, which is then used when calling MsgSend() to specify the destination channel. After the Client sends a message to the Server using MsgSend(), the Client process blocks until the Server receives the message using MsgReceive() and then replies using MsgReply(). MsgReceive() also blocks on the server until a message from a client arrives. The advantages of this procedure are: synchronization between the processes, deterministic behavior, safe data exchange and flow-control. Some disadvantages of this system are: blocking behavior, deadlock potential and higher latency compared to asynchronous IPC.

PART 2: Shared resource synchronization with semaphores

Code:

Reader.cpp:

```
// reader.cpp
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h>

using namespace std;

#define SEM_NAME "/logfile_sem"
#define LOG_FILE "/tmp/shared_log.txt"

int main(void) {
   sem_t *sem;
   FILE *fp;
   char line[256];

   // Open the existing named semaphore
   sem = sem_open(SEM_NAME, 0);
   if (sem == SEM_FAILED) {
      perror("sem_open");
      exit(EXIT_FAILURE);
   }

   for (int i = 0; i < 15; i++) {
      sem_wait(sem); // Lock before reading

      fp = fopen(LOG_FILE, "r");
      if (fp == NULL) {
         perror("fopen");
         sem_post(sem);
         sleep(1);
         continue;
      }

      printf("Reader: Current contents of log file:\n");
      while (fgets(line, sizeof(line), fp)) {
         printf("%s", line);
      }
      fclose(fp);

      printf("Reader: End of file.\n\n");
      sem_post(sem); // Unlock after reading
      usleep(600 * 1000);
      }
```

```
    sem_close(sem);
    sem_unlink(SEM_NAME); // Remove semaphore from system
    printf("Reader: Finished reading and semaphore unlinked.\n");

    return 0;
}
```

Writer.cpp:

```
// writer.cpp
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h>
#include <string.h>

using namespace std;

#define SEM_NAME "/logfile_sem"
#define LOG_FILE "/tmp/shared_log.txt"

int main(void) {
    sem_t *sem;
    FILE *fp;
    char buffer[64];

    // Open or create a named semaphore
    sem = sem_open(SEM_NAME, O_CREAT, 0644, 1);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }

    // Remove content of existing file
    FILE *f = fopen(LOG_FILE, "w");
    fclose(f);

    for (int i = 0; i < 5; i++) {
        sem_wait(sem); // Lock the semaphore before accessing file
        printf("Writer: Writing entry %d...\n", i + 1);

        fp = fopen(LOG_FILE, "a");
        if (fp == NULL) {
            perror("fopen");
            sem_post(sem);
            break;
        }

        // Race Condition DEMO ////////////////////////////////////////////////////////////////////

        // To demo a race condition I split the write into two and added a delay in between
        // COMMENT OUT the semaphores (sem_wait() and sem_post()) to see race condition
```

```
        //snprintf(buffer, sizeof(buffer), "Log entry %d from Writer (PID %d)\n", i + 1, getpid());
        //fputs(buffer, fp);

        // Write PART 1, flush so the reader can see it
        fprintf(fp, "Part 1: Log entry %d \n", i + 1);
        fflush(fp);

        // Delay to create a race window
        usleep(1000 * 1000);  // 1s

        // Write PART 2 and flush again
        fprintf(fp, "Part 2: from Writer (PID %d)\n", getpid());
        fflush(fp);

        fclose(fp);
        ////////////////////////////////////////////////////////////////////////////////////////////

        sem_post(sem); // Release the semaphore
        sleep(1);
    }

    sem_close(sem);
    // Do not unlink here, the reader needs it
    printf("Writer: Finished writing.\n");

    return 0;
}
```

Execution Log:

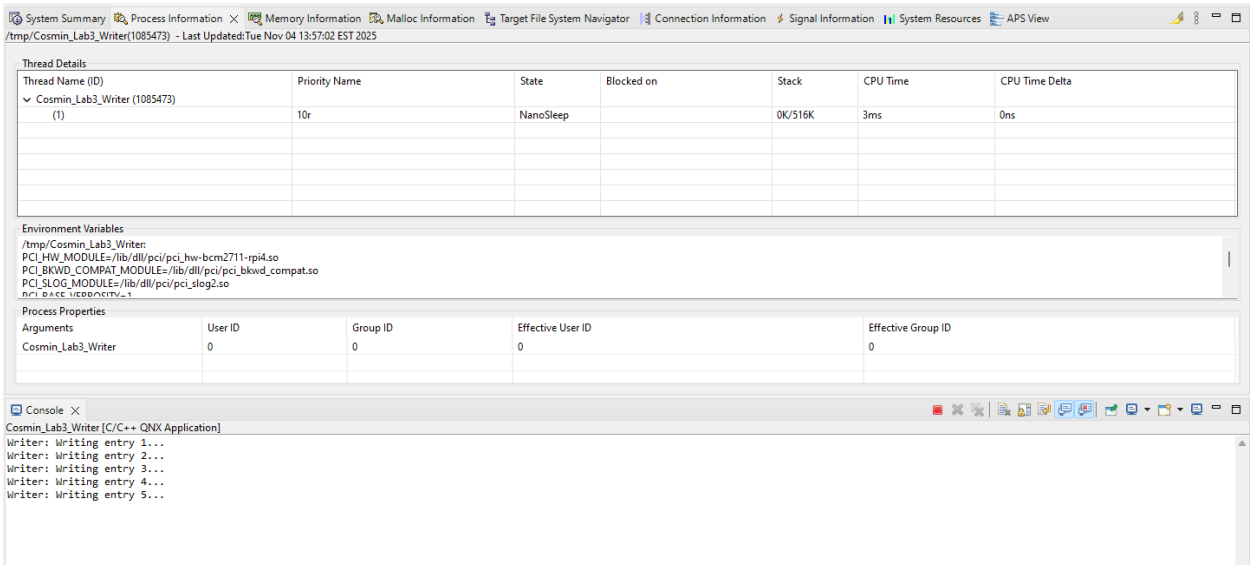


Figure 5: Execution log of Writer writing to logfile.

```
Problems  Tasks  Console  Properties  Terminal
<terminated> Cosmin_Lab3_Reader [C/C++ QNX Application] /tmp/Cosmin_Lab3_Reader on 192.168.141.9 pid 1253412 (11/4/25, 2:58 PM) (Terminated Nov 4, 2025, 2:58:22 PM)
Reader: Current contents of log file:
Part 1: Log entry 1
Part 2: from Writer (PID 1253409)
Part 1: Log entry 2
Part 2: from Writer (PID 1253409)
Reader: End of file.

Reader: Current contents of log file:
Part 1: Log entry 1
Part 2: from Writer (PID 1253409)
Part 1: Log entry 2
Part 2: from Writer (PID 1253409)
Reader: End of file.

Reader: Current contents of log file:
Part 1: Log entry 1
Part 2: from Writer (PID 1253409)
Part 1: Log entry 2
Part 2: from Writer (PID 1253409)
Part 1: Log entry 3
Part 2: from Writer (PID 1253409)
Reader: End of file.

Reader: Current contents of log file:
Part 1: Log entry 1
Part 2: from Writer (PID 1253409)
Part 1: Log entry 2
Part 2: from Writer (PID 1253409)
Part 1: Log entry 3
Part 2: from Writer (PID 1253409)
Reader: End of file.

Reader: Current contents of log file:
Part 1: Log entry 1
Part 2: from Writer (PID 1253409)
Part 1: Log entry 2
Part 2: from Writer (PID 1253409)
Part 1: Log entry 3
Part 2: from Writer (PID 1253409)
Part 1: Log entry 4
Part 2: from Writer (PID 1253409)
Reader: End of file.
```

Figure 6: Execution log of Reader reading from logfile – split writing in 2 parts – with semaphore

I modified the Writer to split each logfile entry into 2 parts, with a delay in between, to demonstrate the mutual exclusion that the semaphore provides. This modification is highlighted in the Code section above. Figure 5 shows the console output of the writer writing to the logfile. Figure 6 shows the Reader reading from the logfile and it is important to note that it reads whole entries (part 1 and part 2 together) thanks to the semaphore providing mutual exclusion when accessing the shared logfile. Also, the logfile contents remain consistent between reads, without any data corruption.

Analysis of Race Condition:

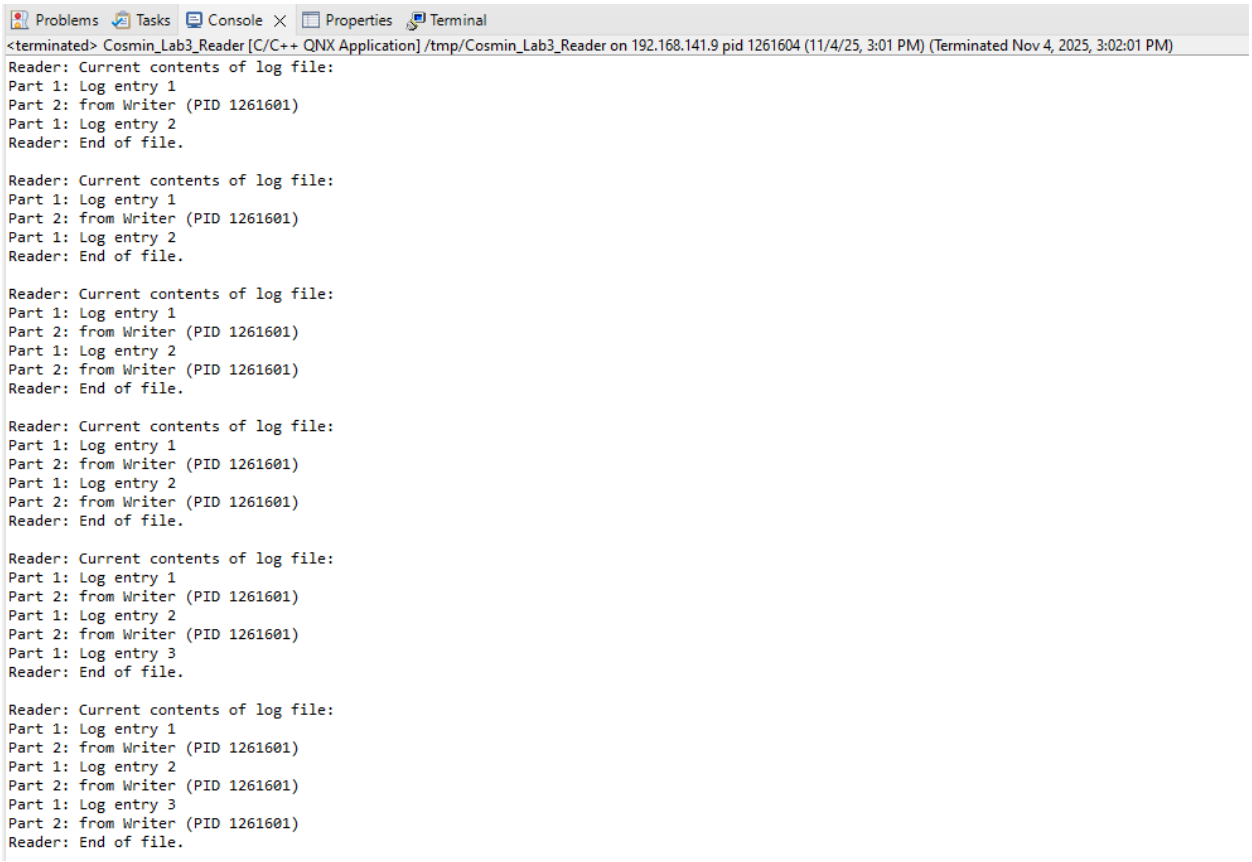


Figure 7: Execution log of Reader reading from logfile – split writing in 2 parts – No semaphore

Figure 7 shows what the Reader is reading when the semaphore isn't used (sem_wait() and sem_post() were commented out). It is clearly visible that the reader sometimes reads half-entries (only one part), which demonstrates race condition. When the semaphore isn't used, the Reader is allowed to read the logfile during the delay period in between part 1 and part 2 entry writes, because there is nothing protecting the access to the logfile. This causes inconsistent reads and can even cause interleaved writes when the Writer tries to write at the same time as a Reader reads. Concurrent logfile access is dangerous and should be avoided in an RTOS.

Analysis of Semaphore Solution:

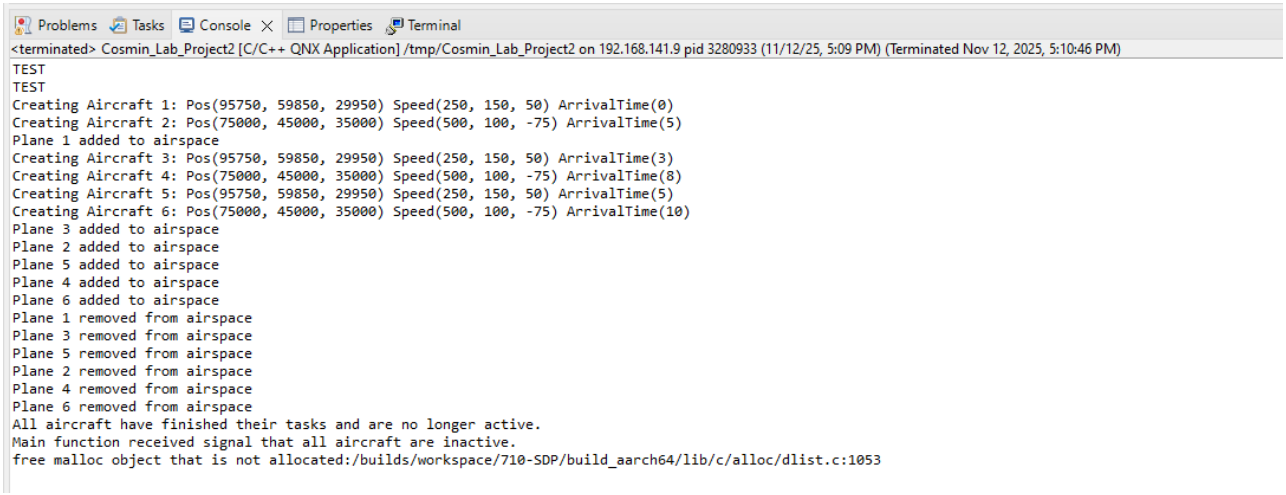
Semaphores are used to synchronize and serialize access to resources across processes because they lock access to specific processes at one time. A mutex also ensures that only one thread can access a resource at one time, but they are confined to threads within the same process, since they are not visible globally in the system. By contrast, a named semaphore has a system-wide name and is managed by the kernel. It is accessible by any process addressing that name and it is very suited to provide deterministic inter-process synchronization.

Discussion:

Semaphore-controlled access endures that only a specific number of processes can access any shared resource at any given time, whereas uncontrolled concurrent access lets many processes access that resource which may interfere and cause race conditions or data corruption. Semaphores help enforce determinism in an RTOS where many processes fight for the same resource by making each process wait until that resource is available, guaranteeing predictable and safe interactions. This is important for sharing resources like hardware devices, system logs or configuration files because data integrity and timing precision must be maintained.

Lab 4 part 1: ATC

Execution logs:



CONCLUSION

To summarize, in this experiment we explored inter-process synchronous messaging and inter-process resource synchronization using semaphores. In the first part we saw how QNX uses synchronized message passing for inter-process communication where a client blocks until the server responds and vice-versa. For the second part of the experiment, we have seen how race conditions can happen if a shared resource (logfile) is accessed concurrently by different processes and then how to lock access to the resource using semaphores so that no race conditions can occur. These concepts are important and very useful for designing and developing real-time systems where tasks are expected to be executed on time with high predictability.