

6CS005
High Performance Computing
Week 5 Workshop

Revision on Multithreading

Student Number: 2408869

Name: Narayan Lohani

Group: L6CG15

Table of Contents

Problem Explanation.....	4
Problematic Output	4
Explanation	5
Busy Wait Source Code	6
Mutex Source Code.....	7
Semaphore Source Code	9
Busy Wait Ooutput.....	11
Mutex Output	12
Semaphore Output	13

Table of Figures

Figure 1: Problematic Output.....	4
Figure 2: Solution using Busy Wait	6
Figure 3: Solution using Mutex [1].....	7
Figure 4: Solution using Mutex [2].....	8
Figure 5: Solution using Semaphore [1]	9
Figure 6: Solution using Semaphore [2]	10
Figure 7: Output Busy Wait	11
Figure 8: Output Mutex	12
Figure 9: Output Semaphore	13

1. The following program demonstrates 3 thread sending string messages to each other, using a global array. The messages are sent meant to be sent in the following order:

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

char *messages[3] = {NULL, NULL, NULL};

void *messenger(void *p)
{
    long tid = (long)p;
    char tmpbuf[100];

    for(int i=0; i<10; i++)
    {
        /* Sending a message */
        long int dest = (tid + 1) % 3;
        sprintf(tmpbuf, "Hello from Thread %ld!", tid);
        char *msg = strdup(tmpbuf);
        messages[dest] = msg;
        printf("Thread %ld sent the message to Thread %ld\n", tid, dest);

        /* Receiving a message */
        printf("Thread %ld received the message '%s'\n", tid,
messages[tid]);
        free(messages[tid]);
        messages[tid] = NULL;
    }
    return NULL;
}

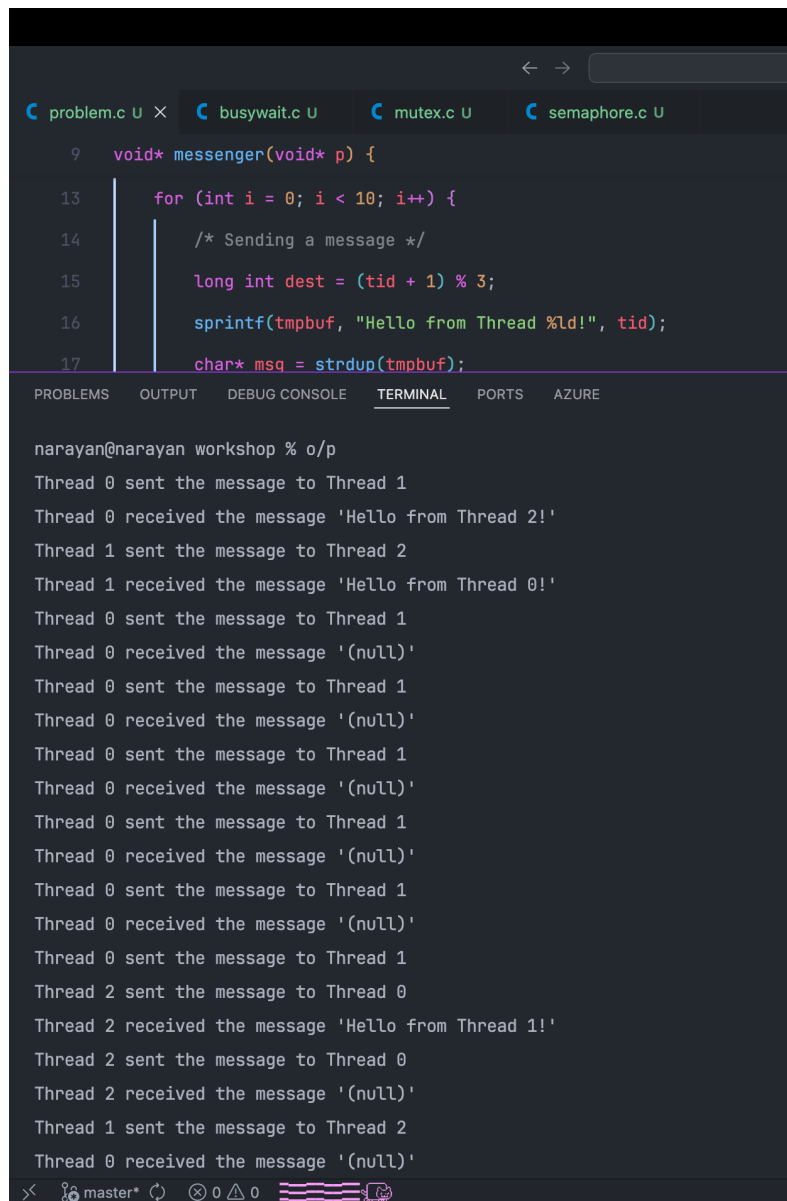
void main()
{
    pthread_t thrID1, thrID2, thrID3;

    pthread_create(&thrID1, NULL, messenger, (void *)0);
    pthread_create(&thrID2, NULL, messenger, (void *)1);
    pthread_create(&thrID3, NULL, messenger, (void *)2);
    pthread_join(thrID1, NULL);
    pthread_join(thrID2, NULL);
    pthread_join(thrID3, NULL);
}
```

Problem Explanation

No, it does not. In the given program, all three threads simultaneously access the shared messages array to send and receive messages. The sections where a thread reads from or writes to the array constitute critical sections, but they are not protected. This lack of protection allows multiple threads to access the same memory at the same time, resulting in race conditions. Consequently, threads may read NULL messages or overwrite messages before they are received, making the program's behavior unpredictable and incorrect.

Problematic Output



```
9 void* messenger(void* p) {
13     for (int i = 0; i < 10; i++) {
14         /* Sending a message */
15         long int dest = (tid + 1) % 3;
16         sprintf(tmpbuf, "Hello from Thread %ld!", tid);
17         char* msg = strdup(tmpbuf);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

narayan@narayan workshop % o/p
Thread 0 sent the message to Thread 1
Thread 0 received the message 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 1 received the message 'Hello from Thread 0!'
Thread 0 sent the message to Thread 1
Thread 0 received the message '(null)'
Thread 0 sent the message to Thread 1
Thread 0 received the message '(null)'
Thread 0 sent the message to Thread 1
Thread 0 received the message '(null)'
Thread 0 sent the message to Thread 1
Thread 0 received the message '(null)'
Thread 0 sent the message to Thread 1
Thread 0 received the message '(null)'
Thread 0 sent the message to Thread 1
Thread 2 sent the message to Thread 0
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 2 received the message '(null)'
Thread 1 sent the message to Thread 2
Thread 0 received the message '(null)'
```

Figure 1: Problematic Output

Explanation

2. Using the technique of “busy-waiting” to correct the program, and establishing the correct order of messages.

This code uses a shared variable flag to control which thread can run at a time. Each thread waits in a loop until flag equals its thread ID, so only one thread enters the critical section at a time. The critical section includes reading and writing messages. By doing this, it prevents multiple threads from accessing the messages array simultaneously, avoiding race conditions. After a thread finishes sending and receiving, it updates flag to allow the next thread to run, ensuring a strict order. This way, messages are always read after they are written, and no thread reads NULL, solving the original synchronization problem.

[Source Code](#)

[Output](#)

3. Use pthread “mutex” to correct the program in (1). You will need multiple mutexes

This code uses one mutex per thread to control which thread can run at a time. Before creating the threads, all mutexes are locked, so no thread can start immediately. After the threads are created, thread 0's mutex is unlocked first, allowing it to enter the critical section. Each thread waits to lock its own mutex before reading from or writing to the shared messages array, ensuring that only one thread accesses the array at a time and preventing race conditions. After finishing sending and receiving, each thread unlocks the next thread's mutex, allowing the next thread to run. This sequential unlocking continues in a turn, ensuring strict order so that messages are always sent before being received and the synchronization problem is solved.

[Source Code](#)

[Output](#)

4. Use semaphores to correct the program in (1).

This code uses one semaphore per thread to control which thread can run at a time. Each thread waits on its own semaphore (sem_wait) before entering the critical section, which includes reading from and writing to the shared messages array. Only the thread whose semaphore is available can run, so multiple threads cannot access the array simultaneously, preventing race conditions. After sending its message, each thread signals the next thread's semaphore (sem_post), allowing the next thread to run. Thread 0 skips receiving on the first iteration using the condition if (!(i == 0 && tid == 0)), so it does not try to read a message that does not exist yet. By passing control sequentially with semaphores, the threads run in a strict order, ensuring that messages are always sent before being received and solving the synchronization problem.

[Source Code](#)

[Output](#)

Busy Wait Source Code

```
hpc_herald - busywait.c

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  int flag = 0;
8  char* messages[3] = { NULL, NULL, NULL };
9
10 void* messenger(void* p) {
11     long tid = (long)p;
12     char tmpbuf[100];
13
14     for (int i = 0; i < 10; i++) {
15         while (flag != tid);
16
17         if (!(i == 0 && tid == 0)) {
18             /* Receiving a message */
19             printf("Thread %ld received the message '%s'\n", tid, messages[tid]);
20             free(messages[tid]);
21             messages[tid] = NULL;
22         }
23
24         /* Sending a message */
25         long int dest = (tid + 1) % 3;
26         sprintf(tmpbuf, "Hello from Thread %ld!", tid);
27         char* msg = strdup(tmpbuf);
28         messages[dest] = msg;
29         printf("Thread %ld sent the message to Thread %ld\n", tid, dest);
30
31         flag = (flag + 1) % 3;
32     }
33     return NULL;
34 }
35
36 int main() {
37     pthread_t thrID1, thrID2, thrID3;
38
39     pthread_create(&thrID1, NULL, messenger, (void*)0);
40     pthread_create(&thrID2, NULL, messenger, (void*)1);
41     pthread_create(&thrID3, NULL, messenger, (void*)2);
42     pthread_join(thrID1, NULL);
43     pthread_join(thrID2, NULL);
44     pthread_join(thrID3, NULL);
45
46     return 0;
47 }
48
```

Figure 2: Solution using Busy Wait

Mutex Source Code

```
hpc_herald - mutex.c

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  #define NUM_THREADS 3
8
9  pthread_mutex_t locker[NUM_THREADS];
10
11 char* messages[3] = { NULL, NULL, NULL };
12
13 void* messenger(void* p) {
14     long tid = (long)p;
15     char tmpbuf[100];
16
17     for (int i = 0; i < 10; i++) {
18         pthread_mutex_lock(&locker[tid]);
19
20         if (!(i == 0 && tid == 0)) {
21             /* Receiving a message */
22             printf("Thread %ld received the message '%s'\n", tid, messages[tid]);
23             free(messages[tid]);
24             messages[tid] = NULL;
25         }
26
27         /* Sending a message */
28         long int dest = (tid + 1) % 3;
29         sprintf(tmpbuf, "Hello from Thread %ld!", tid);
30         char* msg = strdup(tmpbuf);
31         messages[dest] = msg;
32         printf("Thread %ld sent the message to Thread %ld\n", tid, dest);
33
34         long next = (tid + 1) % NUM_THREADS;
35         pthread_mutex_unlock(&locker[next]);
36     }
37     return NULL;
38 }
```

Figure 3: Solution using Mutex [1]

```
hpc_herald - mutex.c

39
40 int main() {
41     pthread_t threads[NUM_THREADS];
42
43     // Mutex initialize
44     for (int i = 0; i < NUM_THREADS; i++)
45         pthread_mutex_init(&locker[i], NULL);
46
47     // Lock all mutexes before execution
48     for (int i = 0; i < NUM_THREADS; i++)
49         pthread_mutex_lock(&locker[i]);
50
51     // Thread creation
52     for (long i = 0; i < NUM_THREADS; i++) {
53         pthread_create(&threads[i], NULL, messenger, (void*)i);
54     }
55
56     // Unlock thread 0 so it starts first
57     pthread_mutex_unlock(&locker[0]);
58
59     // Thread join and Mutex destroy
60     for (int i = 0; i < NUM_THREADS; i++) {
61         pthread_join(threads[i], NULL);
62         pthread_mutex_destroy(&locker[i]);
63     }
64
65     return 0;
66 }
67
```

Figure 4: Solution using Mutex [2]

Semaphore Source Code

```
hpc_herald - semaphore.c

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <pthread.h>
5  #include <unistd.h>
6  #include <semaphore.h>
7
8  #define NUM_THREADS 3
9
10 sem_t sema_locker[NUM_THREADS];
11
12 char* messages[3] = { NULL, NULL, NULL };
13
14 void* messenger(void* p) {
15     long tid = (long)p;
16     char tmpbuf[100];
17
18     for (int i = 0; i < 10; i++) {
19         sem_wait(&sema_locker[tid]);
20
21         if (!(i == 0 && tid == 0)) {
22             /* Receiving a message */
23             while (messages[tid] == NULL) {}
24             printf("Thread %ld received the message '%s'\n", tid, messages[tid]);
25             free(messages[tid]);
26             messages[tid] = NULL;
27         }
28
29         /* Sending a message */
30         long int dest = (tid + 1) % 3;
31         sprintf(tmpbuf, "Hello from Thread %ld!", tid);
32         char* msg = strdup(tmpbuf);
33         messages[dest] = msg;
34         printf("Thread %ld sent the message to Thread %ld\n", tid, dest);
35
36         long next = (tid + 1) % NUM_THREADS;
37         sem_post(&sema_locker[next]);
38     }
39     return NULL;
40 }
```

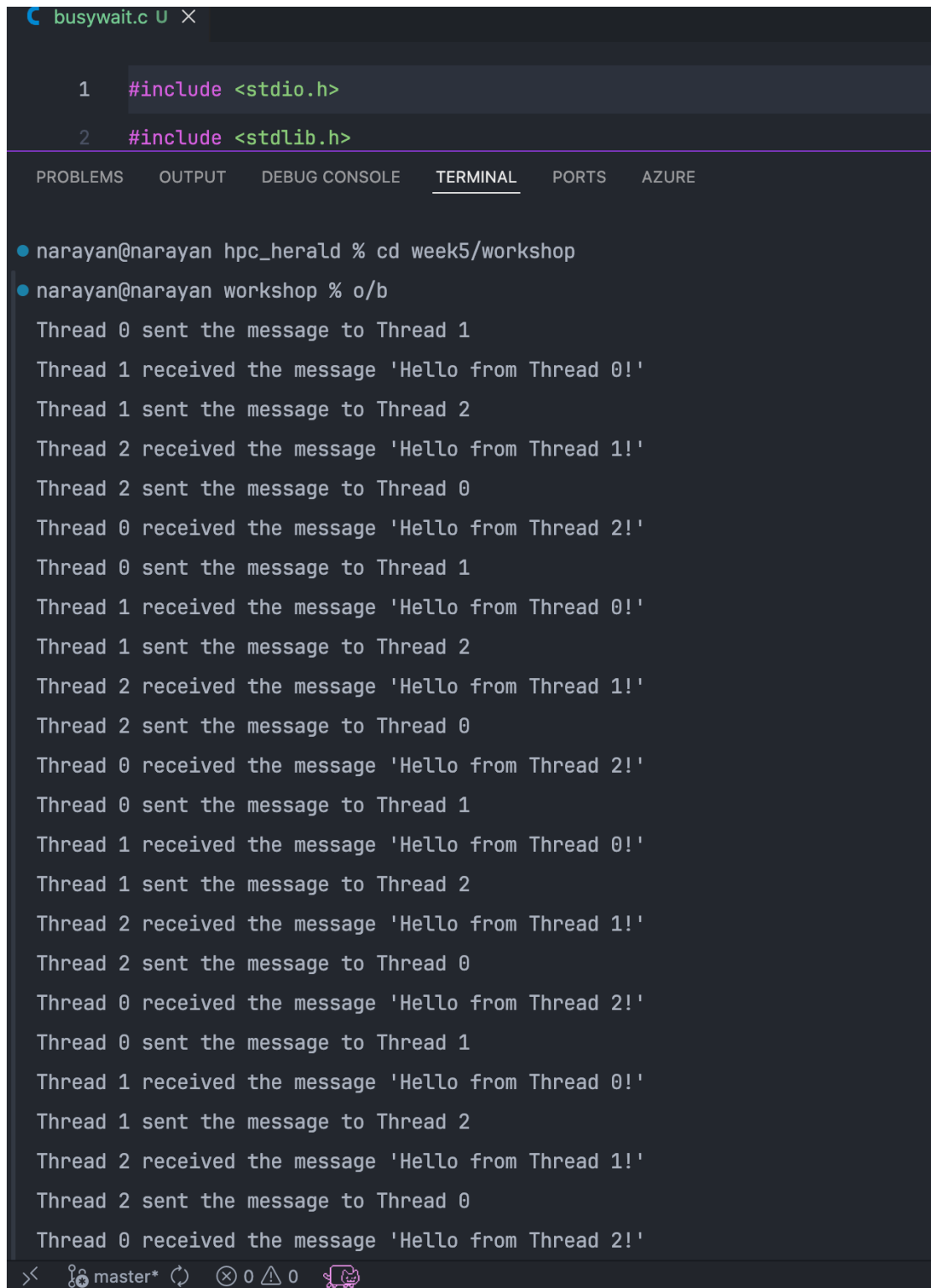
Figure 5: Solution using Semaphore [1]

```
hpc_herald - semaphore.c

41
42  int main() {
43      pthread_t threads[NUM_THREADS];
44
45      sem_init(&sema_locker[0], 0, 1);
46      sem_init(&sema_locker[1], 0, 0);
47      sem_init(&sema_locker[2], 0, 0);
48
49      // Thread creation
50      for (long i = 0; i < NUM_THREADS; i++) {
51          pthread_create(&threads[i], NULL, messenger, (void*)i);
52      }
53
54      // Thread join
55      for (int i = 0; i < NUM_THREADS; i++) {
56          pthread_join(threads[i], NULL);
57      }
58
59      // Destroy Semaphore
60      for (int i = 0; i < NUM_THREADS; i++)
61          sem_destroy(&sema_locker[i]);
62      return 0;
63  }
64
```

Figure 6: Solution using Semaphore [2]

Busy Wait Ooutput



The screenshot shows a code editor with a file named `busywait.c`. The code contains two lines: `#include <stdio.h>` and `#include <stdlib.h>`. Below the code, the `TERMINAL` tab is active, displaying the output of a program. The output shows a sequence of messages between three threads (0, 1, and 2) in a circular fashion. Thread 0 sends a message to Thread 1, Thread 1 sends a message to Thread 2, and Thread 2 sends a message to Thread 0. This cycle repeats five times. The messages are: "Thread 0 sent the message to Thread 1", "Thread 1 received the message 'Hello from Thread 0!'", "Thread 1 sent the message to Thread 2", "Thread 2 received the message 'Hello from Thread 1!'", "Thread 2 sent the message to Thread 0", "Thread 0 received the message 'Hello from Thread 2!'", "Thread 0 sent the message to Thread 1", "Thread 1 received the message 'Hello from Thread 0!'", "Thread 1 sent the message to Thread 2", "Thread 2 received the message 'Hello from Thread 1!'", "Thread 2 sent the message to Thread 0", "Thread 0 received the message 'Hello from Thread 2!'", "Thread 0 sent the message to Thread 1", "Thread 1 received the message 'Hello from Thread 0!'", "Thread 1 sent the message to Thread 2", "Thread 2 received the message 'Hello from Thread 1!'", "Thread 2 sent the message to Thread 0", and "Thread 0 received the message 'Hello from Thread 2!'".

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

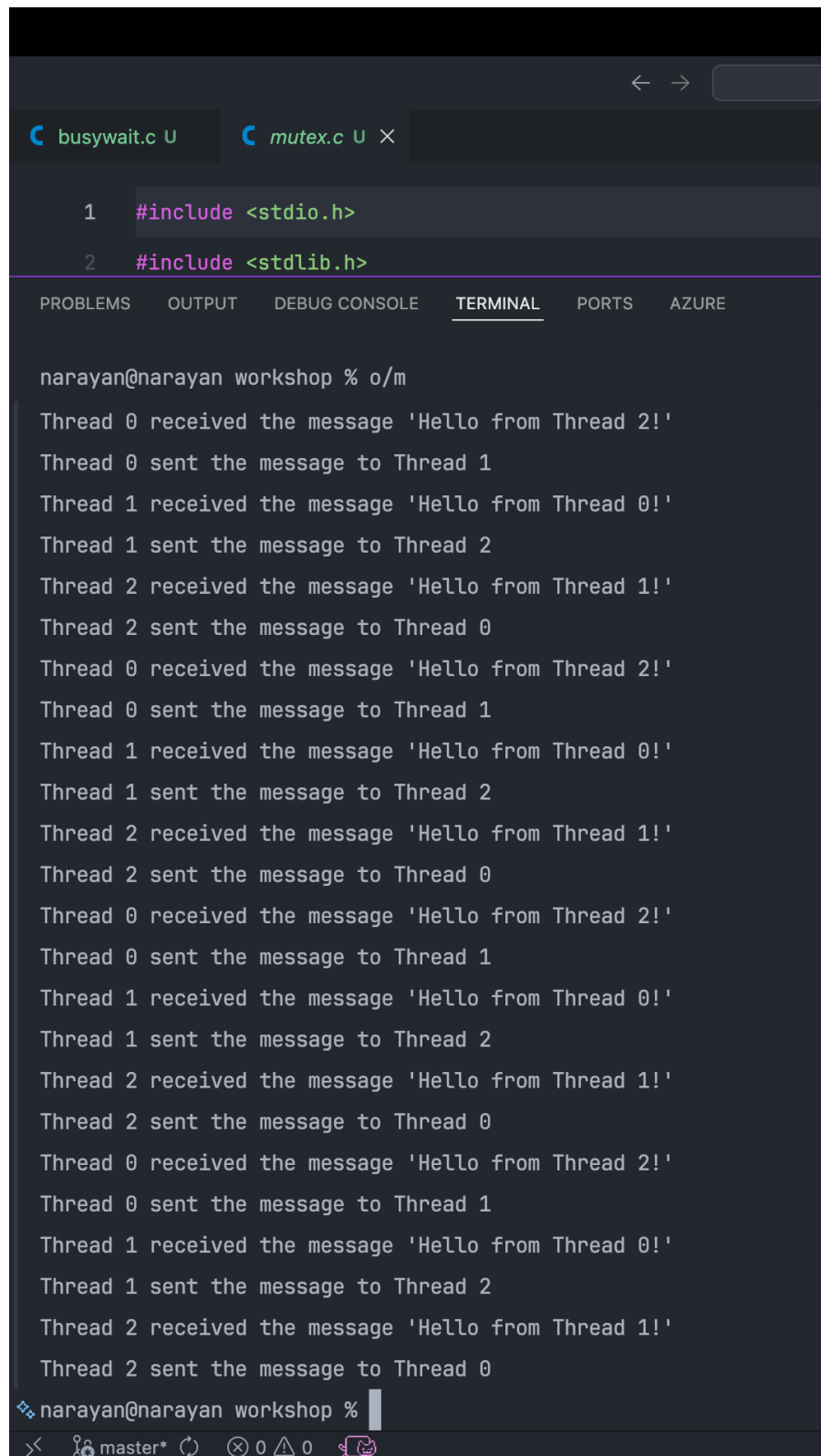
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

```
• narayan@narayan hpc_herald % cd week5/workshop
• narayan@narayan workshop % o/b
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
```

< master* 0 0

Figure 7: Output Busy Wait

Mutex Output



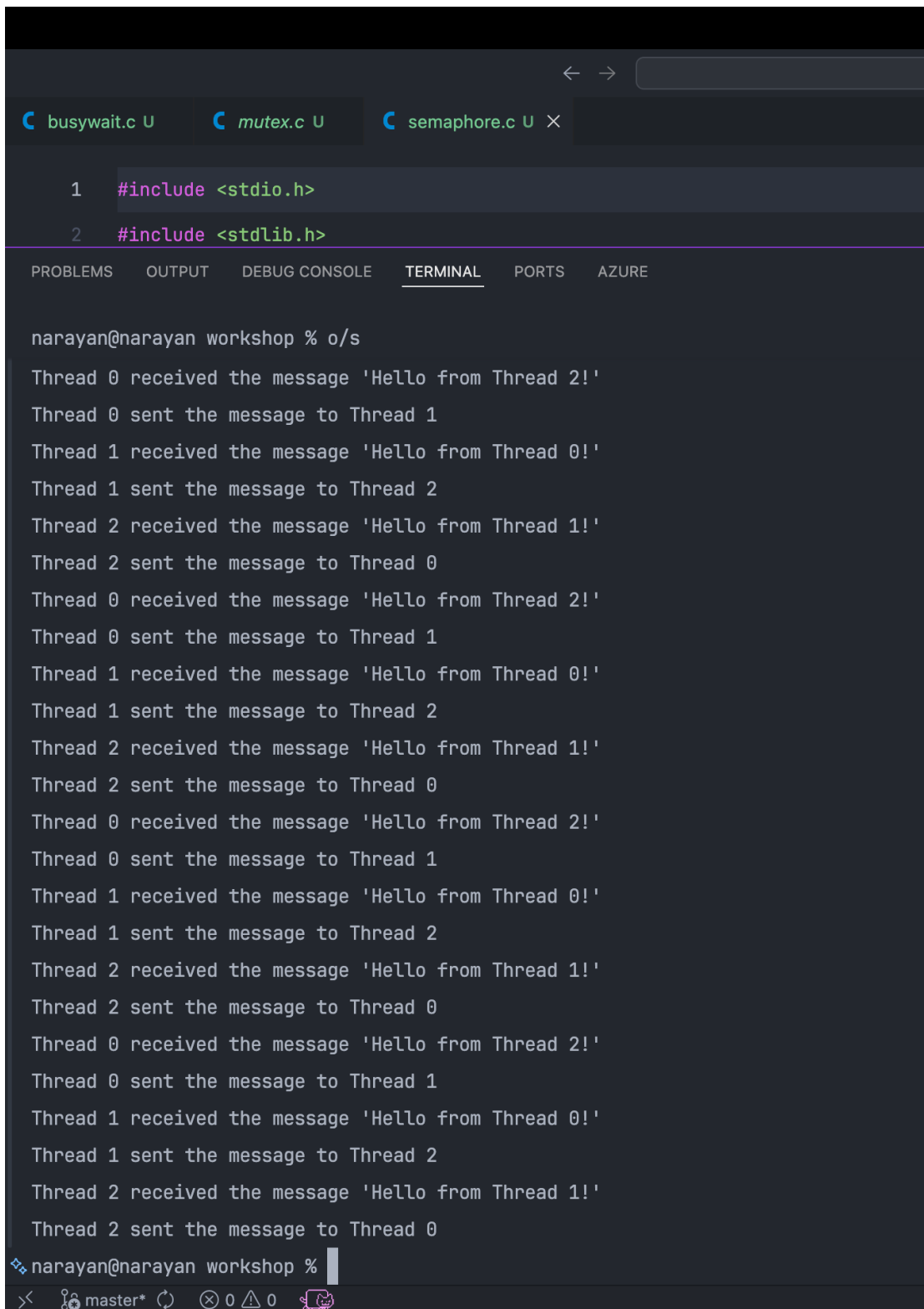
```
1  #include <stdio.h>
2  #include <stdlib.h>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

```
narayan@narayan workshop % o/m
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
narayan@narayan workshop %
```

Figure 8: Output Mutex

Semaphore Output



The screenshot shows a code editor with three tabs: `busywait.c`, `mutex.c`, and `semaphore.c`. The `semaphore.c` tab is active, displaying the following code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

Below the code editor is a terminal window with the following output:

```
narayan@narayan workshop % o/s
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 0 received the message 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 1 received the message 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 2 received the message 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
narayan@narayan workshop %
```

The terminal output shows a sequence of messages between three threads (0, 1, and 2) using a semaphore. The messages are: 'Hello from Thread 2!', 'Hello from Thread 0!', and 'Hello from Thread 1!'. The threads send messages to each other in a cyclic manner: Thread 0 sends to Thread 1, Thread 1 sends to Thread 2, and Thread 2 sends to Thread 0. This pattern repeats multiple times.

Figure 9: Output Semaphore