University of Wolverhampton

School of Mathematics and Computer Science

Student Number: 2407710

Name: Dhadkan K.C.

6CS005 High Performance Computing Week 5 Workshop

POSIX Thread and Semaphores

Tasks – Threads and messaging

You may need to refer to the Week 5 lecture slides in order to complete these tasks.

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:

a. Thread 0 sends Thread 1 a message

b. Thread 1 receives the message

c. Thread 1 sends Thread 2 a message

d. Thread 2 receives the message

e. Thread 2 sends Thread 0 a message

f. Thread 0 receives the message

g. This then repeats from (a) 10 times

week5 > C qn1.c > ...

```c
#include <stdio.h>
#include <stdint.h>

char *messages[3] = {NULL, NULL, NULL};
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond[3];
intptr_t turn = 0;

void *messenger(void *p)
{
    intptr_t tid = (intptr_t)p;
    intptr_t dest = (tid + 1) % 3;
    char tmpbuf[100];

    for (int i = 0; i < 10; i++)
    {
        pthread_mutex_lock(&lock);

        while (turn != tid)
            pthread_cond_wait(&cond[tid], &lock);

        sprintf(tmpbuf, "Hello from Thread %ld!", (long)tid);
        messages[dest] = strdup(tmpbuf);
        printf("Thread %ld sent the message to Thread %ld\n", (long)tid, (long)dest);

        printf("Thread %ld received: '%s'\n", (long)tid, messages[tid]);
        free(messages[tid]);
        messages[tid] = NULL;

        turn = dest;
        pthread_cond_signal(&cond[turn]);

        pthread_mutex_unlock(&lock);
    }

    return NULL;
}

int main()
{
    pthread_t thr[3];

    for (int i = 0; i < 3; i++)
        pthread_cond_init(&cond[i], NULL);

    pthread_create(&thr[0], NULL, messenger, (void*)0);
    pthread_create(&thr[1], NULL, messenger, (void*)1);
    pthread_create(&thr[2], NULL, messenger, (void*)2);

    pthread_join(thr[0], NULL);
    pthread_join(thr[1], NULL);
    pthread_join(thr[2], NULL);

    return 0;
}
```

```
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent the message to Thread 1
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 0 sent the message to Thread 1
Thread 0 received: 'Hello from Thread 2!'
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 0 sent the message to Thread 1
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
Thread 2 received: 'Hello from Thread 1!'
Thread 0 sent the message to Thread 1
Thread 0 sent the message to Thread 1
Thread 0 received: 'Hello from Thread 2!'
Thread 0 received: 'Hello from Thread 2!'
Thread 1 sent the message to Thread 2
Thread 1 received: 'Hello from Thread 0!'
Thread 2 sent the message to Thread 0
Thread 2 received: 'Hello from Thread 1!'
PS C:\Clzstuffs\HPC\week5>
```

Explanation:

The program works by making sure only one thread runs its send-and-receive step at a time. A shared variable called turn decides which thread is allowed to act. Each thread waits until turn matches its own ID, then it sends a message to the next thread, receives the message meant for itself, and updates turn so the next thread can continue. Condition variables wake up the correct thread at the right time, and a mutex protects the shared messages so threads don't interfere with each other. This guarantees the threads pass messages in the correct order without any race conditions.

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

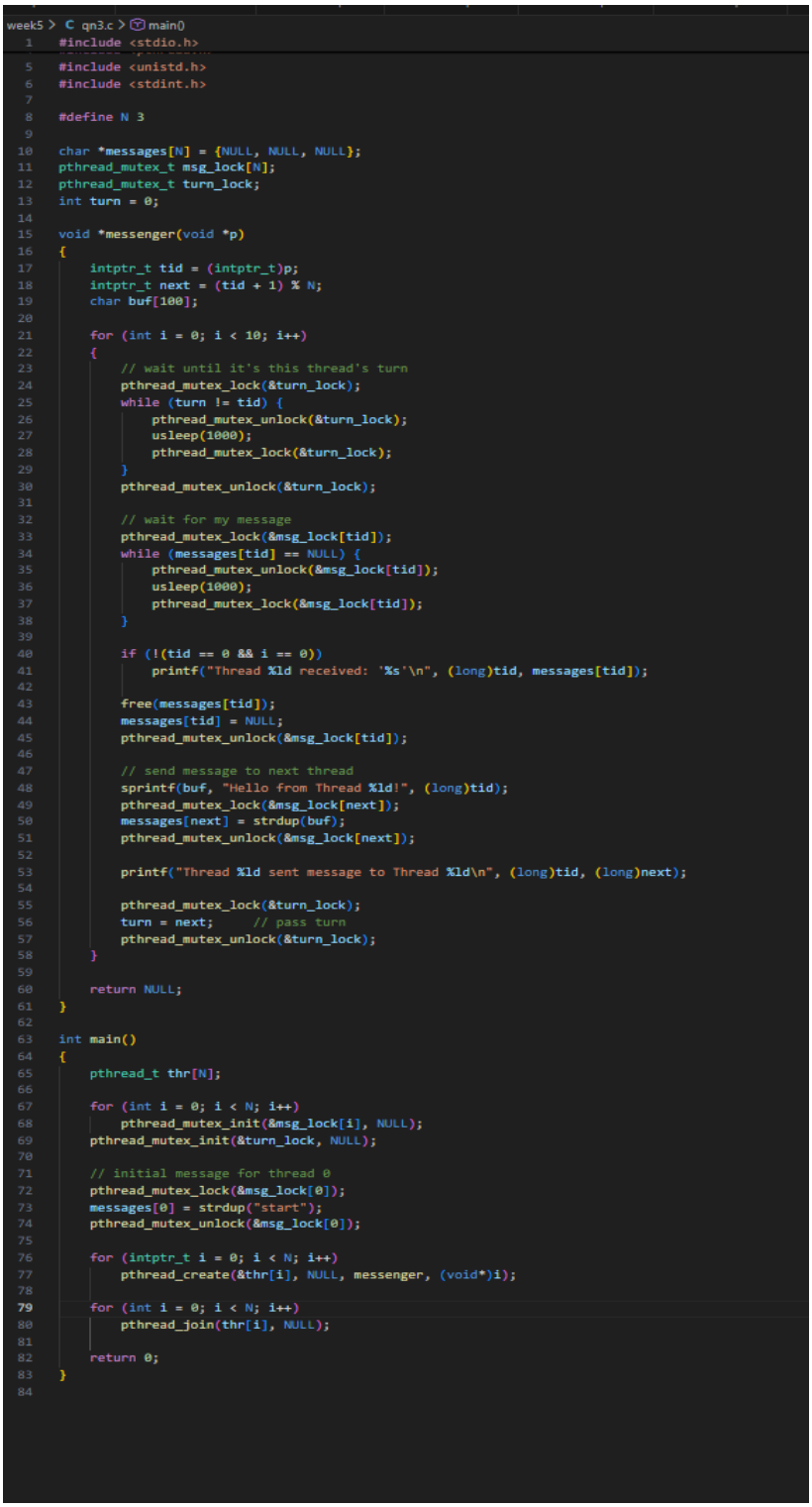
week5 > C qn2.c > messenger(void *)

```c
char *messages[3] = {NULL, NULL, NULL};
volatile int turn = 0;   // controls order of execution

void *messenger(void *p) {
    intptr_t tid = (intptr_t)p;
    intptr_t next = (tid + 1) % 3;
    char buf[100];

    for (int i = 0; i < 10; i++) {

        // Wait until it's THIS thread's turn
        while (turn != tid)
            usleep(1000);

        // Wait until a message is available for this thread
        while (messages[tid] == NULL)
            usleep(1000);

        // Do not print the very first message for Thread 0
        if (!(tid == 0 && i == 0))
            printf("Thread %ld received: '%s'\n", (long)tid, messages[tid]);

        free(messages[tid]);
        messages[tid] = NULL;

        // Create outgoing message
        sprintf(buf, "Hello from Thread %ld!", (long)tid);
        messages[next] = strdup(buf);

        printf("Thread %ld sent message to Thread %ld\n", (long)tid, (long)next);

        // Pass control to the next thread
        turn = next;
    }

    return NULL;
}

int main() {
    pthread_t thr[3];

    // Initial message to start thread 0
    messages[0] = strdup("start");

    for (intptr_t i = 0; i < 3; i++)
        pthread_create(&thr[i], NULL, messenger, (void*)i);

    for (int i = 0; i < 3; i++)
        pthread_join(thr[i], NULL);

    return 0;
}
```

Explanation: Each thread created by the program runs the function messenger. Inside this thread routine, every thread passes message to the next thread and reads their own message as well. At first, each thread retrieves its thread id and calculates the next thread that will receive its message. The thread routine enter a loop that executes 10 rounds of sending and receiving messages. Before each thread can act in each round, it must wait for two conditions: • Must be their turn (turn == tid) • A message must already exist for it (messages[tid] != NULL) The thread waits using a while loop with usleep() to avoid high CPU usage and once both conditions are satisfied, the thread receives a message from the previous thread. It prints the message received, frees the memory that held it and clears its slot in the message array. The only

exception is thread 0 on the first iteration, which receives a dummy starting message and therefore skips printing. After receiving its message, the thread sends a new message to the next thread by writing a formatted string into a temporary buffer and storing a supplicated copy using strdup() into messages[next]. It then prints that it sent the message. To continue the cycle, the thread hands over control by updating the turn global variable to the next thread id. This allows that thread to proceed and keeps the message passing order synchronized. In the main function, an array of thread identifiers is declared. To allow thread 0 to begin immediately, the main function places an initial “starting message” into message[0]. This ensures thread 0 does not block on its first iteration, since the thread routine requires each thread to already have a message waiting for it. Then, three threads are created in a loop passing each thread its ID (0,1,2) as an argument. Each thread then executes the messenger function. After all threads are are done processing, the main function waits for them to complete by calling pthread_join() on each one. This allows the program to not exit before all the threads finish their work.

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



week5 > C qn3.c > main()

```c
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>

#define N 3

char *messages[N] = {NULL, NULL, NULL};
pthread_mutex_t msg_lock[N];
pthread_mutex_t turn_lock;
int turn = 0;

void *messenger(void *p)
{
    intptr_t tid = (intptr_t)p;
    intptr_t next = (tid + 1) % N;
    char buf[100];

    for (int i = 0; i < 10; i++)
    {
        // wait until it's this thread's turn
        pthread_mutex_lock(&turn_lock);
        while (turn != tid) {
            pthread_mutex_unlock(&turn_lock);
            usleep(1000);
            pthread_mutex_lock(&turn_lock);
        }
        pthread_mutex_unlock(&turn_lock);

        // wait for my message
        pthread_mutex_lock(&msg_lock[tid]);
        while (messages[tid] == NULL) {
            pthread_mutex_unlock(&msg_lock[tid]);
            usleep(1000);
            pthread_mutex_lock(&msg_lock[tid]);
        }

        if (!(tid == 0 && i == 0))
            printf("Thread %ld received: '%s'\n", (long)tid, messages[tid]);

        free(messages[tid]);
        messages[tid] = NULL;
        pthread_mutex_unlock(&msg_lock[tid]);

        // send message to next thread
        sprintf(buf, "Hello from Thread %ld!", (long)tid);
        pthread_mutex_lock(&msg_lock[next]);
        messages[next] = strdup(buf);
        pthread_mutex_unlock(&msg_lock[next]);

        printf("Thread %ld sent message to Thread %ld\n", (long)tid, (long)next);

        pthread_mutex_lock(&turn_lock);
        turn = next;      // pass turn
        pthread_mutex_unlock(&turn_lock);
    }

    return NULL;
}

int main()
{
    pthread_t thr[N];

    for (int i = 0; i < N; i++)
        pthread_mutex_init(&msg_lock[i], NULL);
    pthread_mutex_init(&turn_lock, NULL);

    // initial message for thread 0
    pthread_mutex_lock(&msg_lock[0]);
    messages[0] = strdup("start");
    pthread_mutex_unlock(&msg_lock[0]);

    for (intptr_t i = 0; i < N; i++)
        pthread_create(&thr[i], NULL, messenger, (void*)i);

    for (int i = 0; i < N; i++)
        pthread_join(thr[i], NULL);

    return 0;
}
```

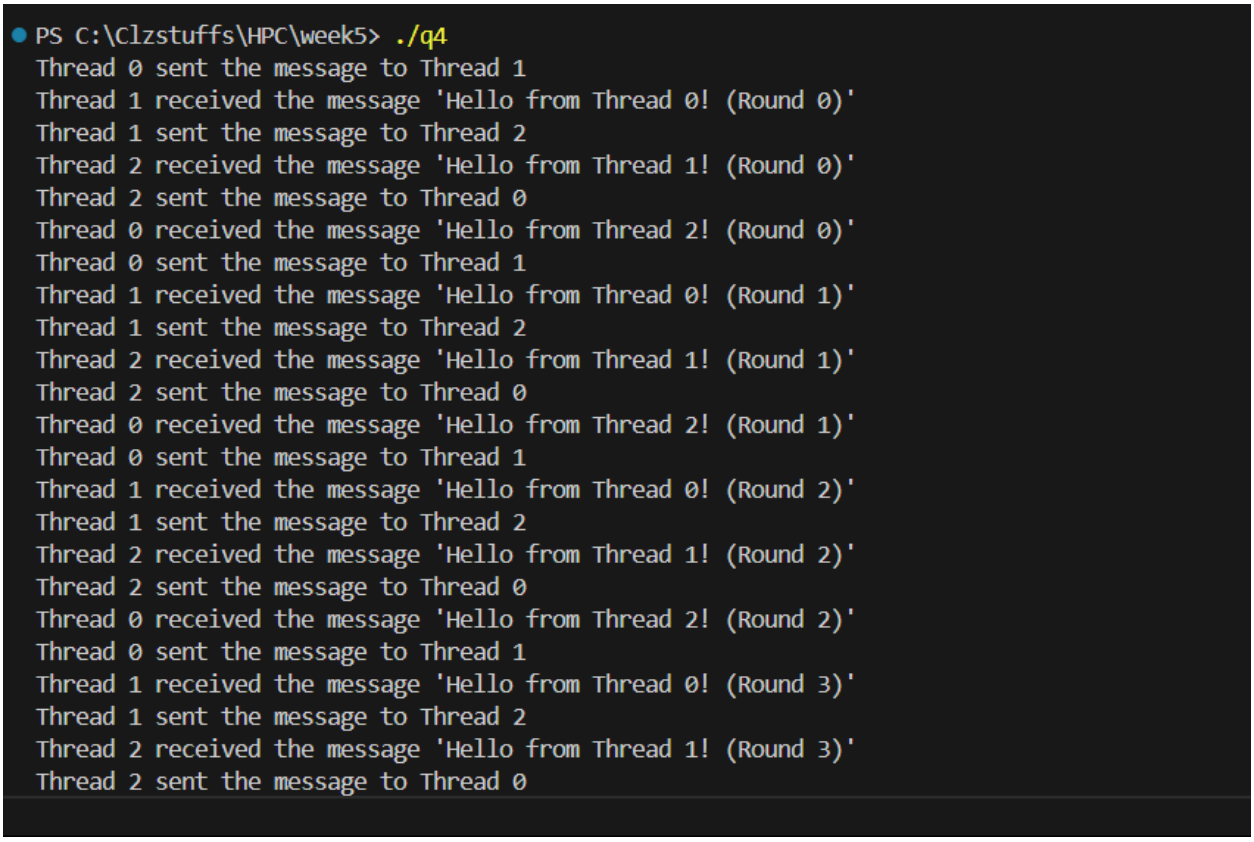
```
PS C:\Clzstuffs\HPC\week5> ./qn3
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent message to Thread 0
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent message to Thread 0
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent message to Thread 0
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent message to Thread 0
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
Thread 2 sent message to Thread 0
Thread 0 received: 'Hello from Thread 2!'
Thread 0 sent message to Thread 1
Thread 1 received: 'Hello from Thread 0!'
Thread 1 sent message to Thread 2
Thread 2 received: 'Hello from Thread 1!'
```

Explanation: Each thread runs the messenger() function, where it participates in a circular message-passing routine. At the start, it determines its ID and which thread should receive its next message. The routine then enters a loop that runs for ten rounds.

In each round, the thread first locks the shared mutex to safely check shared state. It then waits on the condition variable until two requirements are met: it must be that thread's turn to act, and there must already be a message waiting for it. This waiting uses pthread_cond_wait(), which puts the thread to sleep efficiently instead of consuming CPU like busy-waiting. Once the conditions are satisfied, the thread receives its message, prints it, and frees the memory used. After that, it prepares and sends a new message to the next thread in the ring. Finally, it updates the turn variable to pass control to the next thread, broadcasts a wake-up signal to other threads via the condition variable, releases the mutex and continues to the next iteration. The main() function begins by printing a startup message and inserting an initial "Start message" into the slot for thread 0 so that it can begin the first round immediately. It then creates three threads, each running the messenger() routine with its ID passed as an argument. After launching them, main() waits for all three threads to complete their ten rounds by calling pthread_join() on each thread. This ensures the message-passing cycle finishes cleanly before the program exits. Once all threads have terminated, the program prints a final message and ends normally.

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

```c
#include <stdint.h>

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

void *messenger(void *p)
{
    intptr_t tid = (intptr_t)p;
    int next = (tid + 1) % 3;
    char tmpbuf[100];

    for (int i = 0; i < 10; i++)
    {
        sem_wait(&sem[tid]);

        if (!(tid == 0 && i == 0))
        {
            printf("Thread %ld received the message '%s'\n", (long)tid, messages[tid]);
            free(messages[tid]);
            messages[tid] = NULL;
        }

        sprintf(tmpbuf, "Hello from Thread %ld! (Round %d)", (long)tid, i);
        messages[next] = strdup(tmpbuf);
        printf("Thread %ld sent the message to Thread %d\n", (long)tid, next);

        sem_post(&sem[next]);
    }

    return NULL;
}

int main()
{
    pthread_t threads[3];

    sem_init(&sem[0], 0, 1);
    sem_init(&sem[1], 0, 0);
    sem_init(&sem[2], 0, 0);

    messages[0] = strdup("start");

    for (intptr_t i = 0; i < 3; i++)
        pthread_create(&threads[i], NULL, messenger, (void*)i);

    for (int i = 0; i < 3; i++)
        pthread_join(threads[i], NULL);

    return 0;
}
```



Explanation: Each thread runs the messenger() function, which participates in a circular message-passing routine among three threads. Semaphores are used to control which thread is allowed to run at any moment. Each thread begins by calling sem_wait() on its own semaphore which blocks the thread until another thread signals it is its turn. Once unblocked, the thread receives a message left for it by the previous thread, prints it, frees the memory, and clears its slot. It then prepares a new message for the next thread in the ring, stores it in the shared message array, and prints that it has sent the message. Finally, the thread uses sem_post() to wake the next thread, allowing the circular communication to continue. This pattern repeats for 10 rounds, ensuring strict ordering without busy-waiting. The main() function initializes the program by creating three semaphores one for each thread and setting all of them to zero so no thread can run initially. It then places an initial message into messages[0] and signals sem_post(&sem[0]), giving thread 0 permission to start the message ring. After that, it creates the three threads, allowing each to execute the messenger() routine. Since only thread 0 has been unblocked, it runs first and then passes control to thread 1, which then passes to thread 2, continuing in a loop. When all threads finish their 10 rounds, the main function joins them and prints a final completion message.