OPERATING SYSTEM ASSIGNMENT 3

SAILORS CONUNDRUM

A SYNCHRONIZATION PROBLEM

GROUP DETAILS:

SERIAL NUMBER	REGISTRATION NUMBER	ROLL NUMBER	NAME
1	150905041	09	RAAVISHU SANGHVI
2	150905254	43	FARAZ FARUQI
3	150905112	23	UTKARSH AGARWAL
4	150905288	50	SAMRAT DUTTA
5	150905107	22	KUNAL CHHAJED

INTRODUCTION

The Sailor’s Conundrum is a synchronization problem that involves four threads: an Agent and three Sailors. The sailors loop forever, first waiting for the resources, then getting those resources and at last sailing. The three resources that the sailor needs are a ship, a crew and money. We assume that the Agent has abundant supply of all the three resources and each sailor has unlimited supply of any one of the three resources. Hence one sailor has a ship, another one has a crew and the last one has unlimited money.

The agent then randomly chooses two resources and makes them available for the sailors to use. Depending on which resources are available the sailor that has the complementary resource chooses the other two resources and sets for sail.

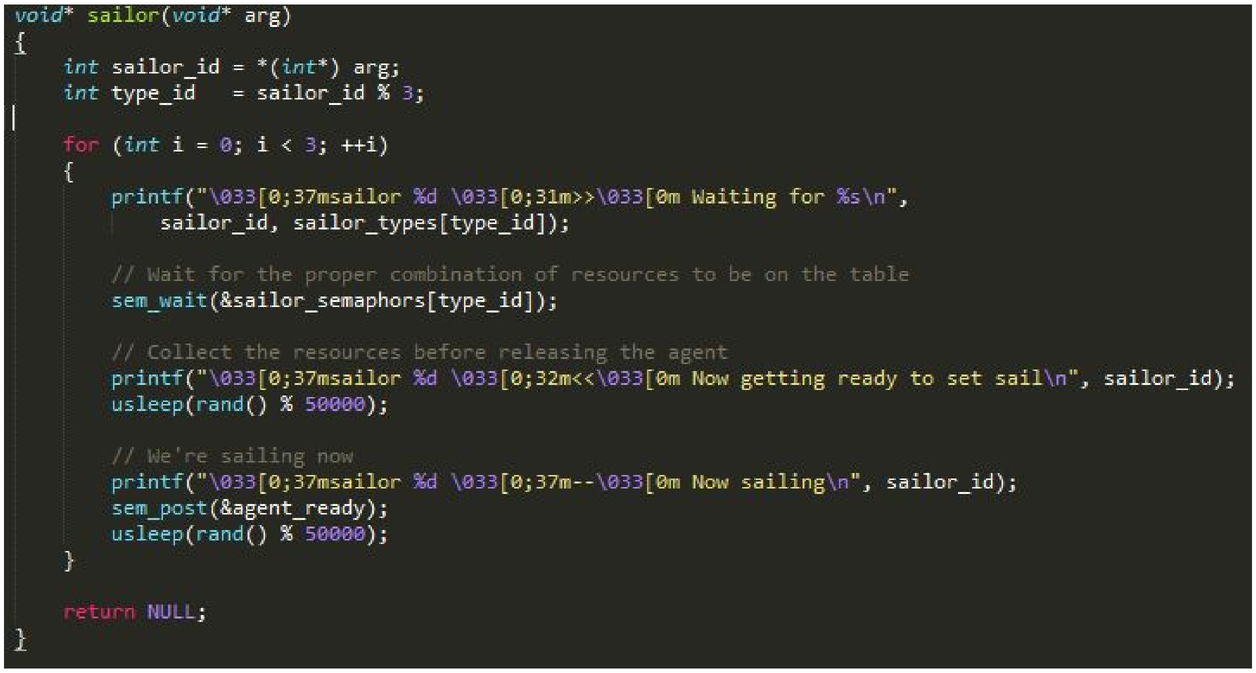
To explain the following problem in terms of Operating System, we can see that the Agent acts/represents an Operating System that allocates resources and the sailors represent various applications that need the various resources. The problem is to make sure that if resources are available that would allow an application to proceed then that particular application should be woken up. This can be achieved by using the concept of semaphores and exclusive mutex.

METHODOLOGY

First we create a semaphore for the agent and then we create three semaphores for the sailors. Initially there are no resources available for the sailors except those resources that are always available to a sailor.

One way to see this problem is that when the Agent runs, it creates two pushers. Each pusher gets one resource i.e. either a ship or a crew or money, and puts the resources in one room with other resources. Due to synchronization each pusher comes to the room where there are three sailors and checks the already present resources. If a complete set of resources can be assembled, then the pusher gives the resources to the respective sailor, and the sailor sets of to sail. If not, then the pusher leaves without giving the resources to anyone.

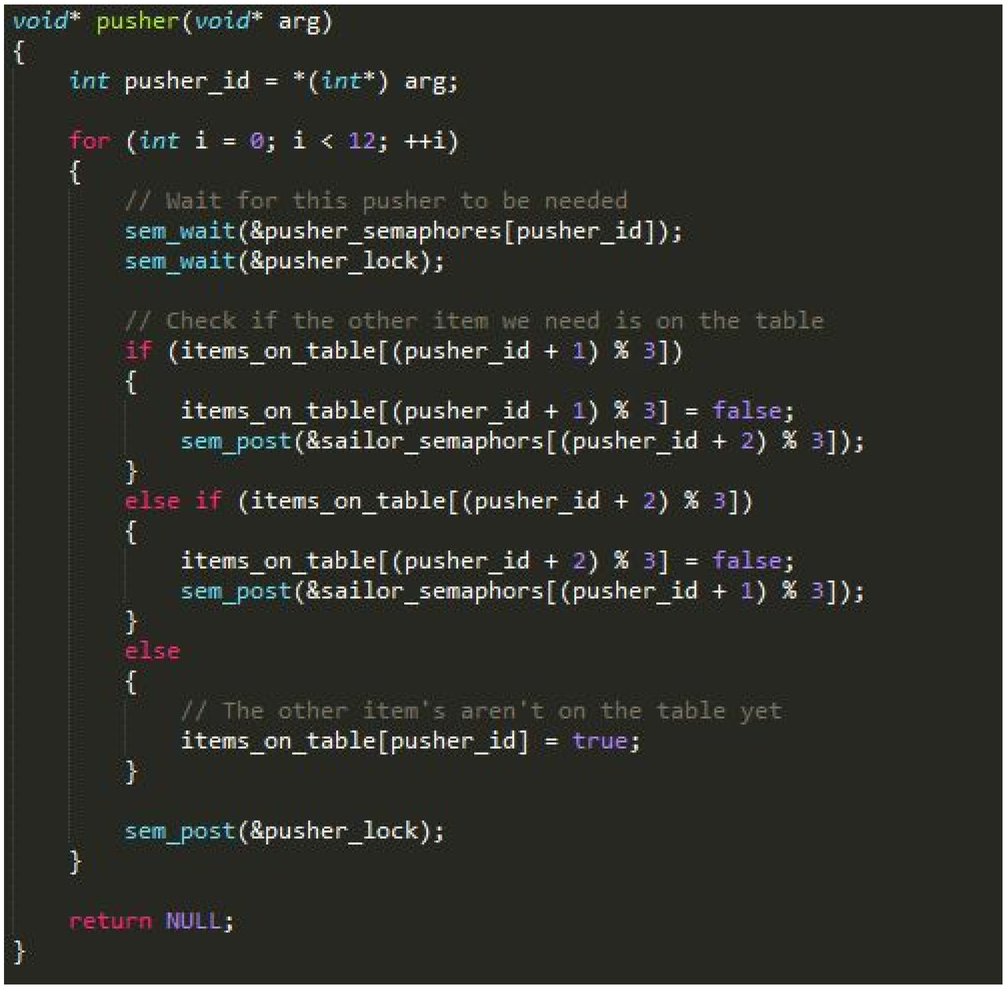
SAILOR



```c
void* sailor(void* arg)
{
    int sailor_id = *(int*) arg;
    int type_id   = sailor_id % 3;

    for (int i = 0; i < 3; ++i)
    {
        printf("\033[0;37msailor %d \033[0;31m>>\033[0m Waiting for %s\n",
            sailor_id, sailor_types[type_id]);

        // Wait for the proper combination of resources to be on the table
        sem_wait(&sailor_semaphors[type_id]);

        // Collect the resources before releasing the agent
        printf("\033[0;37msailor %d \033[0;32m<<\033[0m Now getting ready to set sail\n", sailor_id);
        usleep(rand() % 50000);

        // We're sailing now
        printf("\033[0;37msailor %d \033[0;37m--\033[0m Now sailing\n", sailor_id);
        sem_post(&agent_ready);
        usleep(rand() % 50000);
    }

    return NULL;
}
```

The function sailor gets an argument from the main through the v=create function. This argument is then used by the variable type_id. The modulus function is used to assign any one of the three sailor types i.e. “ship and money”, “ship and crew” or “money or crew” which have the values 0,1 and 2 respectively. The sem_wait function is then used on the sailor with that particular type_id. The sem_wait function is used here to signify that the sailor has taken the resources and will be going on a sail shortly. Once the sailor starts the sail, the sem_post(&agent_ready) function is used to signify that the agent has given the resources to that particular sailor and now won’t be giving any more resources till the sailor returns.

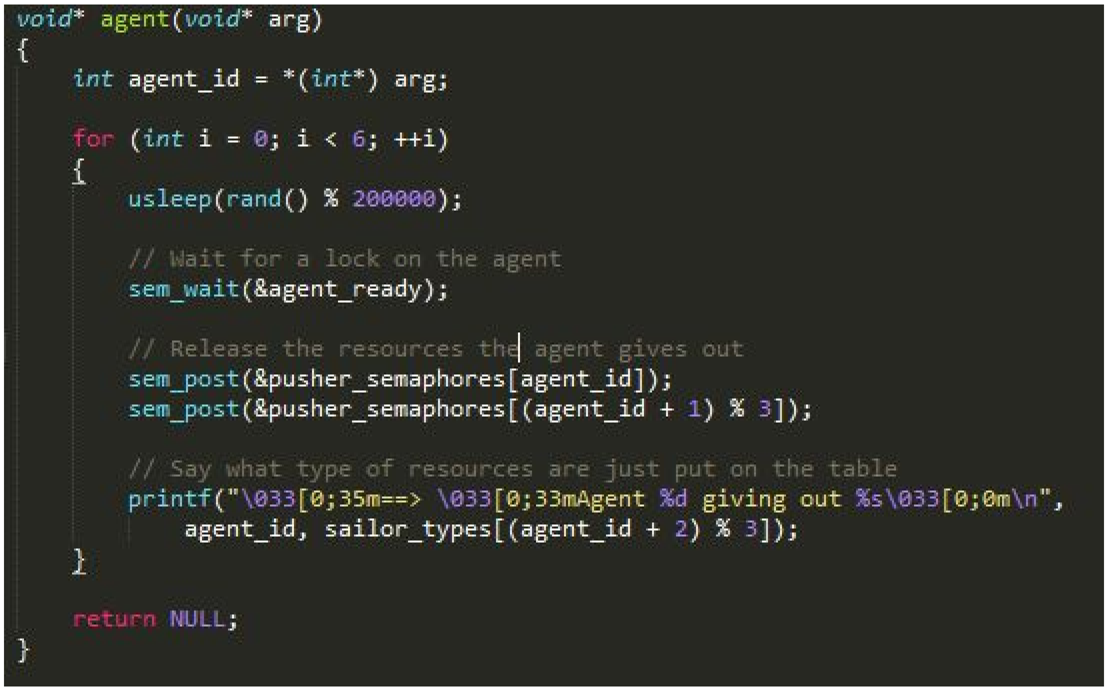
PUSHER



```
void* pusher(void* arg)
{
    int pusher_id = *(int*) arg;

    for (int i = 0; i < 12; ++i)
    {
        // Wait for this pusher to be needed
        sem_wait(&pusher_semaphores[pusher_id]);
        sem_wait(&pusher_lock);

        // Check if the other item we need is on the table
        if (items_on_table[(pusher_id + 1) % 3])
        {
            items_on_table[(pusher_id + 1) % 3] = false;
            sem_post(&sailor_semaphors[(pusher_id + 2) % 3]);
        }
        else if (items_on_table[(pusher_id + 2) % 3])
        {
            items_on_table[(pusher_id + 2) % 3] = false;
            sem_post(&sailor_semaphors[(pusher_id + 1) % 3]);
        }
        else
        {
            // The other item's aren't on the table yet
            items_on_table[pusher_id] = true;
        }

        sem_post(&pusher_lock);
    }

    return NULL;
}
```

As seen in the sailor function, even the pusher function gets an argument which is used as the pusher_id. Once a pusher_id is selected it is then made to wait using the two sem_wait functions. Then the pusher function checks which other resources are available. If the other resources are available then the item_on_table value for that particular resource is made false as to signify that the resources needed can be taken by the sailor. But if no other resources are available then the item_on_table value that particular resource is made true. Once this is done the pusher_lock is then released using the sem_post(&pusher_lock) command.

AGENT



```
void* agent(void* arg)
{
    int agent_id = *(int*) arg;

    for (int i = 0; i < 6; ++i)
    {
        usleep(rand() % 200000);

        // Wait for a lock on the agent
        sem_wait(&agent_ready);

        // Release the resources the agent gives out
        sem_post(&pusher_semaphores[agent_id]);
        sem_post(&pusher_semaphores[(agent_id + 1) % 3]);

        // Say what type of resources are just put on the table
        printf("\033[0;35m==> \033[0;33mAgent %d giving out %s\033[0;0m\n",
            agent_id, sailor_types[(agent_id + 2) % 3]);
    }

    return NULL;
}
```

The parameter passed is used to know the agent_id. Once the agent is selected we use the same_wait() function to lock the agent. Then the sem_post() functions are used to release the resources that agent is currently giving out to the sailors. With that the agent also specifies the type of resources that are being given out.

We've also used some animation to depict the whole scenario to make the project more interactive. The animation works with respect to the code running in the background.

CODE

```
#include <pthread.h>

#include <semaphore.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <vlc/vlc.h>


// matches ship

// tobacco money

// paper crew

// An agent semaphore represents items on the table

sem_t agent_ready;


// Each sailor semaphore represents when a sailor has the items they need

sem_t sailor_semaphors[3];


// This is an array of strings describing what each sailor type needs

char* sailor_types[3] = { "ship & money", "ship & crew", "money & crew" };


// This list represents item types that are on the table. This should correspond

// with the sailor_types, such that each item is the one the sailor has. So the
// first item would be paper, then tobacco, then matches.
bool items_on_table[3] = { false, false, false };

// Each pusher pushes a certian type item, manage these with this semaphore
sem_t pusher_semaphores[3];

int sound(char* arg)
{
        libvlc_instance_t *inst;
        libvlc_media_player_t *mp;
        libvlc_media_t *m;

        // load the engine
        inst = libvlc_new(0, NULL);

        // create a file to play
        m = libvlc_media_new_path(inst, arg);

        // create a media play playing environment
        mp = libvlc_media_player_new_from_media(m);

        // release the media now.
        libvlc_media_release(m);

        // play the media_player
        libvlc_media_player_play(mp);

        sleep(3); // let it play for 10 seconds
```

```
            // stop playing
            libvlc_media_player_stop(mp);

            // free the memory.
            libvlc_media_player_release(mp);

            libvlc_release(inst);



            return 0;
}
/**
 * sailor function, handles waiting for the item's that they need, and then
 * smoking. Repeat this three times
 */
void* sailor(void* arg)
{
   int sailor_id = *(int*) arg;
   int type_id   = sailor_id % 3;
   int random;
   // Smoke 3 times
   for (int i = 0; i < 3; ++i)
   {
             printf("\033[0;37msailor %d \033[0;31m>>\033[0m Waiting for %s\n",
                          sailor_id, sailor_types[type_id]);

             // Wait for the proper combination of items to be on the table
             sem_wait(&sailor_semaphors[type_id]);


             // Make the ship before releasing the agent
             printf("\033[0;37msailor %d \033[0;32m<<\033[0m Now Making ship\n", sailor_id);
             sound("construction.mp3");
             // random = rand() % 10000000;


             // printf("%d\n", random);


             usleep(2000000);


            // We're sailing now
             printf("\033[0;37msailor %d \033[0;37m--\033[0m Now sailing\n", sailor_id);
            sound("water.mp3");
            // random = rand() % 10000000;
            // printf("%d\n",random);


            usleep(2000000);
            sem_post(&agent_ready);


   }


   return NULL;
}


// This semaphore gives the pusher exclusive access to the items on the table
sem_t pusher_lock;
```

```
/**
 * The pusher is responsible for releasing the proper sailor semaphore when the
 * right item's are on the table.
 */
void* pusher(void* arg)
{
   int pusher_id = *(int*) arg;

   for (int i = 0; i < 12; ++i)
   {
        // Wait for this pusher to be needed
        sem_wait(&pusher_semaphores[pusher_id]);
        sem_wait(&pusher_lock);

        // Check if the other item we need is on the table
        if (items_on_table[(pusher_id + 1) % 3])
        {
            items_on_table[(pusher_id + 1) % 3] = false;
            sem_post(&sailor_semaphors[(pusher_id + 2) % 3]);
        }
        else if (items_on_table[(pusher_id + 2) % 3])
        {
            items_on_table[(pusher_id + 2) % 3] = false;
            sem_post(&sailor_semaphors[(pusher_id + 1) % 3]);
        }
        else
        {
            // The other item's aren't on the table yet
			items_on_table[pusher_id] = true;
		}

		sem_post(&pusher_lock);
	}

	return NULL;
}

/**
 * The agent puts items on the table
 */
void* agent(void* arg)
{
	int agent_id = *(int*) arg;

	for (int i = 0; i < 6; ++i)
	{
		//usleep(200000);

		// Wait for a lock on the agent
		sem_wait(&agent_ready);

		// Say what type of items we want to put on the table
		printf("\033[0;35m==> \033[0;33mAgent %d giving out %s\033[0;0m\n",
			agent_id, sailor_types[(agent_id + 2) % 3]);

		// Release the items this agent gives out
```

```
            sem_post(&pusher_semaphores[agent_id]);

            sem_post(&pusher_semaphores[(agent_id + 1) % 3]);

   }


   return NULL;

}




/**

 * The main thread handles the agent's arbitration of items.

 */

int main(int argc, char* arvg[])

{

   // Seed our random number since we will be using random numbers

   srand(time(NULL));


   // There is only one agent semaphore since only one set of items may be on

   // the table at any given time. A values of 1 = nothing on the table

   sem_init(&agent_ready, 0, 1);


   // Initalize the pusher lock semaphore

   sem_init(&pusher_lock, 0, 1);


   // Initialize the semaphores for the sailors and pusher

   for (int i = 0; i < 3; ++i)

   {

            sem_init(&sailor_semaphors[i], 0, 0);
```

```
        sem_init(&pusher_semaphores[i], 0, 0);
}



// sailor ID's will be passed to the threads. Allocate the ID's on the stack
int sailor_ids[6];

pthread_t sailor_threads[6];

// Create the 6 sailor threads with IDs
for (int i = 0; i < 6; ++i)
{
        sailor_ids[i] = i;

        if (pthread_create(&sailor_threads[i], NULL, sailor, &sailor_ids[i]) == EAGAIN)
        {
                perror("Insufficient resources to create thread");
                return 0;
        }
}

// Pusher ID's will be passed to the threads. Allocate the ID's on the stack
int pusher_ids[6];

pthread_t pusher_threads[6];

for (int i = 0; i < 3; ++i)
```

```
{
        pusher_ids[i] = i;

        if (pthread_create(&pusher_threads[i], NULL, pusher, &pusher_ids[i]) == EAGAIN)
        {
                perror("Insufficient resources to create thread");
                return 0;
        }
}

// Agent ID's will be passed to the threads. Allocate the ID's on the stack
int agent_ids[3];

pthread_t agent_threads[3];

for (int i = 0; i < 3; ++i)
{
        agent_ids[i] =i;

        if (pthread_create(&agent_threads[i], NULL, agent, &agent_ids[i]) == EAGAIN)
        {
                perror("Insufficient resources to create thread");
                return 0;
        }
}

// Make sure all the sailors are done smoking
for (int i = 0; i < 6; ++i)
```

```
   {
          pthread_join(sailor_threads[i], NULL);
   }
   return 0;
}
```

CONCLUSION & RESULT

We attempted solving the sailor’s conundrum problem with many procedures. First we attempted solving the problem normally, then we tried using mutex to solve the issue of synchronization that occurred in the first attempt. The best results were achieved once we used semaphores in the code. Semaphores helped us synchronize the code perfectly. Semaphores let us halt and resume a particular object when needed, so that there is no clash between any two objects/functions. Hence for any synchronization problem, the best results can be achieved by using semaphores.