|  |  |
| --- | --- |
|  |  |
| Operating Systems |  |
|  |  |
|  | Hodaya Revah: 204596480 |
|  | Gai Gariba: 302592456 |

***HaifaPort.c***

נמל חיפה הוא פרוסס המייצר טרדים מסוג כלי שייט ,מדמה מעבר שלהם בתעלה לנמל אילת (פרוסס אותו הוא יוצר) על ידי שימוש בצינורות וניהול חזרתם לחיפה.

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_RAND\_S

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <windows.h>

#include <time.h>

#define BUFFER\_SIZE 50

#define STRING\_SIZE 256

#define MAX\_VASEALS 50

#define MIN\_VASEALS 2

//Every Vessel waits random time between 5 and 3000 msec between actions

#define MAX\_SLEEP\_TIME 3000 //Miliseconds (3 second)

#define MIN\_SLEEP\_TIME 5 //Miliseconds

#define True 1

#define False 0

//single mutex to protect access to the global state data. Recall, like in lecture notes,

//global state data must be accessed exclusively by each Vessels!

HANDLE mutexPipe;

HANDLE mutexConsole;

//pointer to HANDLE for Array of Scheduling Constraints Semaphores - one for Each Vessels

HANDLE\* sem;

HANDLE StdinRead, StdinWrite; /\* pipe for writing parent to child HaifaPort -> EilatPort\*/

HANDLE StdoutRead, StdoutWrite; /\* pipe for writing child to parent EilatPort -> HaifaPort\*/

DWORD read, written;

// Array holding Philosopher IDS - exclusive for each Philosopher to be sent exclusively to each

int\* VesselsID; // Philosopher's Thread function - to prevent Race Conditions!

DWORD ThreadID; //dummy - for Thread Create invokations

HANDLE\* Vessels; //pointer of Vessels Thread Handles!

SYSTEMTIME lt;//system time paramter to holds time data

int VessIN; //count Vessels that has returned to Haifa

char buffer[BUFFER\_SIZE];//buffer for holding ReadFile receiving value and WriteFile sending value

unsigned int tempRand;//to hold random number from generator

//Thread function for each Vessels

DWORD WINAPI Vessel(PVOID);

//to Initialise and clean global data (mainly for creating the Semaphore Haldles

//before all Threads start and closing them properly after all Threads finish!)

int initGlobalData(int);

void cleanupGlobalData(int);

//function decleration

void StartSailing(int);

void CrossToEilat(int);

int calcSleepTime();

void Anchorage(int);

const char\* getTimeString();

void PrintToConsole(char\*);

int main(int argc, char \*argv[])

{

//check command line argument not empty

if (argc != 2) {

printf("Error :: No command Line Arguments were provided!");

exit(0);

}

//convert command line argument to int,for argument different from int it will return zero

int numOfVessels = atoi(argv[1]);

//validates the numOfVessels value is within the allowable range

if (numOfVessels < MIN\_VASEALS || numOfVessels >MAX\_VASEALS)

{

printf("Error :: invalid value for the number of vessels ");

exit(0);

}

STARTUPINFO si;

PROCESS\_INFORMATION pi;

BOOL success;

TCHAR ProcessName[256];

VessIN = 0;

int index;

char stringToPrint[STRING\_SIZE];

mutexConsole = CreateMutex(NULL, FALSE, TEXT("ConsoleMutex"));

if (mutexConsole == NULL)

{

return False;

}

/\* set up security attributes so that pipe handles are inherited \*/

SECURITY\_ATTRIBUTES sa = { sizeof(SECURITY\_ATTRIBUTES), NULL,TRUE };

/\* allocate memory \*/

ZeroMemory(&pi, sizeof(pi));

/\* create the pipe for writing from parent to child \*/

if (!CreatePipe(&StdinRead, &StdinWrite, &sa, 0)) {

fprintf(stderr, "Create Pipe Failed\n");

return 1;

}

// Ensure the write handle to the pipe for StdinWrite is not inherited.

if (!SetHandleInformation(StdinWrite, HANDLE\_FLAG\_INHERIT, 0))

fprintf(stderr, "StdinWrite SetHandleInformation\n");

/\* create the pipe for writing from child to parent \*/

if (!CreatePipe(&StdoutRead, &StdoutWrite, &sa, 0)) {

fprintf(stderr, "Create Pipe Failed\n");

return 1;

}

// Ensure the read handle to the pipe for STDOUT is not inherited.

if (!SetHandleInformation(StdoutRead, HANDLE\_FLAG\_INHERIT, 0))

fprintf(stderr, "StdoutRead SetHandleInformation\n");

/\* establish the START\_INFO structure for the child process \*/

GetStartupInfo(&si);

si.hStdError = GetStdHandle(STD\_ERROR\_HANDLE);

/\* redirect the standard input to the read end of the pipe \*/

si.hStdOutput = StdoutWrite;

si.hStdInput = StdinRead;

si.dwFlags = STARTF\_USESTDHANDLES;

wcscpy(ProcessName, L"EilatPort.exe");

//wcscpy(ProcessName, L"..//..//EilatPort//Debug//EilatPort.exe");

// Start the child process.

if (!CreateProcess(NULL, // No module name (use command line).

ProcessName, // Command line.

NULL, // Process handle not inheritable.

NULL, // Thread handle not inheritable.

TRUE, // inherit handles .

0, // No creation flags.

NULL, // Use parent's environment block.

NULL, // Use parent's starting directory.

&si, // Pointer to STARTUPINFO structure.

&pi) // Pointer to PROCESS\_INFORMATION structure.

)

{

printf("CreateProcess failed (%d).\n", GetLastError());

return -1;

}

printf("[%s] HaifaPort : Eilat Port connected\n",getTimeString());

printf("[%s]HaifaPort: Confirmation Request for %d Vessels \n", getTimeString(),numOfVessels);

/\* the parent now wants to write to the pipe \*/

if (!WriteFile(StdinWrite, argv[1], BUFFER\_SIZE, &written, NULL))

fprintf(stderr, "Error writing to pipe\n");

/\* now read from the pipe \*/

success = ReadFile(StdoutRead, buffer, BUFFER\_SIZE, &read, NULL);

if (!success) {

fprintf(stderr, "HaifaPort: Error reading from pipe\n");

return -1;

}

/\* close the unused ends of the pipe \*/

CloseHandle(StdoutWrite);

CloseHandle(StdinRead);

if (strcmp(buffer, "FALSE") == 0)

{

printf("HaifaPort: Request didn't approved ,number of vaseals unvalid! \n");

return -1;

}

printf("[%s] HaifaPort: Request approved , Continue... \n",getTimeString());

if (initGlobalData(numOfVessels) == False)

{

printf("HaifaPort: Error while initiating global data \n");

return -1;

}

// creat Vessel threads with VesselId unique grather then 1.

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

{

VesselsID[i] = i+1;

Vessels[i] = CreateThread(NULL, 0, Vessel, &VesselsID[i], 0, &ThreadID);

}

//read from pipe until VessIN is equal to numOfVessel

while (VessIN < numOfVessels)

{

// now have the child read from the pipe

success = ReadFile(StdoutRead, buffer, BUFFER\_SIZE, &read, NULL);;

// we have to output to stderr as stdout is redirected to the pipe

if (!success)

{

fprintf(stderr, "HaifaPort: error reading from pipe , error %d \n",GetLastError());

return -1;

}

else if((index = atoi(buffer))!=0)

{

sprintf(stringToPrint, "[%s] Vessel %s - exiting Canal: Red Sea ==> Med. Sea \n", getTimeString(), buffer);

PrintToConsole(stringToPrint);

if (!ReleaseSemaphore(sem[index-1], 1, NULL))

printf("HaifaPort ::Unexpected error sem[%d].V()\n", index);

VessIN++;

}

}

WaitForMultipleObjects(numOfVessels,Vessels,TRUE,INFINITE);

sprintf(stringToPrint, "[%s] Haifa Port : All Vessels Threads are done \n", getTimeString());

PrintToConsole(stringToPrint);

/\* close the read end of the pipe \*/

/\* close the write end of the pipe \*/

CloseHandle(StdinWrite);

CloseHandle(StdoutRead);

/\* wait for the child to exit \*/

WaitForSingleObject(pi.hProcess, INFINITE);

fprintf(stderr, "[%s] Haifa Port : Exiting... \n", getTimeString());

cleanupGlobalData(numOfVessels);

/\* close all handles \*/

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

return 0;

}

//The Thread Function for each Vessel, gets the Vessl Unique ID (index of the Thread in Array of Threads within the main program)

//This Unique Thread ID (called VessID) is used by the state DB to keep track of each Vessel state at any given time!

DWORD WINAPI Vessel(PVOID Param)

{

//Get the Unique ID from Param and keep it in a local variable

int VesselID = \*(int\*)Param;

StartSailing(VesselID);

Sleep(calcSleepTime());

CrossToEilat(VesselID);

Sleep(calcSleepTime());

Anchorage(VesselID);

return 0;

}

//generic function to randomise a Sleep time between 1 and MAX\_SLEEP\_TIME msec

int calcSleepTime()

{

errno\_t err;

int calc = 0;

while (calc < MIN\_SLEEP\_TIME)

{

err = rand\_s(&tempRand);

if (err != 0)

{

printf\_s("The rand\_s function failed!\n");

}

calc = tempRand % MAX\_SLEEP\_TIME + 1;

}

return calc;

}

//StartSailing function for Vessels VessID - begin to sail

void StartSailing(int vessID)

{

char stringToPrint[STRING\_SIZE];

sprintf(stringToPrint, "[%s] Vessel %d starts sailing @ Haifa Port \n", getTimeString(), vessID);

PrintToConsole(stringToPrint);

}

//CrossToEilat function for Vessels VessID - entering Canal

void CrossToEilat(int vessID)

{

char message[BUFFER\_SIZE];

char stringToPrint[STRING\_SIZE];

//Access to state DB is done exclusively (protected by mutex)

//mutex.P()

WaitForSingleObject(mutexPipe, INFINITE);

sprintf(message, "%d", vessID);

sprintf(stringToPrint, "[%s] Vessel %d - entering Canal: Med. Sea ==> Red Sea \n", getTimeString(), vessID);

PrintToConsole(stringToPrint);

Sleep(calcSleepTime());

/\* the parent now wants to write to the pipe \*/

if (!WriteFile(StdinWrite, message, BUFFER\_SIZE, &written, NULL))

fprintf(stderr, "Error writing to pipe \n %d \n",GetLastError());

//mutex.V() - release mutex

if (!ReleaseMutex(mutexPipe))

fprintf(stderr, "HaifaPort -> CrossToEilat::Unexpected error mutex.V(), error num - %d\n", GetLastError());

//sem[vessID-1].P() - Down on Scheduling Constraint Semaphore for Vessel VessID.

//if test was successful for it (i.e. it started eating), then sem[vessID-1].P() is successfull and Vessel proceeds

//otherwise, sem[vessID-1].P() puts Vessel on Semaphore Waiting queue, until signaled by a pipe indication, Vessel that

//finished Sailing!

WaitForSingleObject(sem[vessID-1], INFINITE);

}

//Anchorage function for Vessel VessID - done sailing

void Anchorage(int VessID)

{

char stringToPrint[STRING\_SIZE];

sprintf(stringToPrint, "[%s] Vessel %d done sailing @ Haifa Port \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

}

//return const string of current locl time

const char\* getTimeString()

{

GetLocalTime(&lt);

static char currentLocalTime[20];

sprintf(currentLocalTime, "%02d:%02d:%02d", lt.wHour, lt.wMinute, lt.wSecond);

return currentLocalTime;

}

//safe console printing

void PrintToConsole(char \*string)

{

WaitForSingleObject(mutexConsole, INFINITE);

fprintf(stderr, "%s", string);

if (!ReleaseMutex(mutexConsole))

printf("PrintToConsole::Unexpected error mutex.V()\n");

}

//Initialise global Semaphores

//If all Successful - return True, otherwise (if problem arises) return False

//This is invoked before all Vessels Threads start running

int initGlobalData(int size)

{

mutexPipe = CreateMutex(NULL, FALSE, NULL);

if (mutexPipe == NULL)

{

return False;

}

Vessels = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, size \* sizeof(HANDLE));

if (Vessels == NULL)

{

printf("initGlobalData:: Error allocating Vessels\n");

return False;

}

sem = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, size\*sizeof(HANDLE));

if (sem == NULL)

{

printf("initGlobalData:: Error allocating sem\n");

return False;

}

VesselsID = malloc(size \* sizeof(int));

if (VesselsID == NULL)

{

printf("initGlobalData:: Error allocating VesselsID\n");

return False;

}

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

{

sem[i] = CreateSemaphore(NULL, 0, 1, NULL);

if (sem[i] == NULL)

{

return False;

}

}

return True;

}

//Close all global semaphore handlers and free dynamic allocations - after all Vessels Threads finish.

void cleanupGlobalData(int size)

{

CloseHandle(mutexPipe);

CloseHandle(mutexConsole);

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

{

CloseHandle(Vessels[i]);

CloseHandle(sem[i]);

}

free(VesselsID);

HeapFree(GetProcessHeap(),HEAP\_NO\_SERIALIZE,sem);

HeapFree(GetProcessHeap(), HEAP\_NO\_SERIALIZE, Vessels);

}

***EilatPort.c***

### נמל אילת פרוסס שמקבל מס' כלי שיט לאישור במידה ומאשר אותם הוא מאזין לכניסת כלי שיט (שמיוצגים כטרדים במערכת), כל כלי שיט שמגיע עובר במחסום, שמשמש כנקודת סנכרון וכשמתקיימים התנאים לשחרור המחסום כלי השיט מתקדמים ונכנסים לרציף הפריקה.

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_RAND\_S

#include <stdio.h> //for I/O

#include <stdlib.h> //for rand/strand

#include <windows.h> //for Win32 API

#include <time.h> //for using the time as the seed to strand!

#include <string.h>

#include <malloc.h>

#include <ctype.h>

#define BUFFER\_SIZE 50

#define STRING\_SIZE 256

#define True 1

#define False 0

//Every Vessel waits random time between 5 and 3000 msec between actions

#define MAX\_SLEEP\_TIME 3000 //Miliseconds (3 second)

#define MIN\_SLEEP\_TIME 5 //Miliseconds

//Every crain random carriege weight

#define MAX\_WEIGHT 50

#define MIN\_WEIGHT 5

//#define getrandom(min, max) (SHORT)((rand() % (int)(((max)+1) - \(min))) + (min))

DWORD ThreadID; //dummy - for Thread Create invokations

HANDLE\* Vessels; //pointer to HANDLE ,after dynamicly allocation will hold threads array of Vessels

int\* VesselsID; //pointer to int ,after dynamicly allocation will hold threads-ID array

int VesselSize; //number of predicted vessel

int VessIN,VessOut; //number of entered vessels so far

HANDLE ReadHandle, WriteHandle;

DWORD read, written;

HANDLE mutexPipe,mutex;//mutexPipe to manage pipe writing, mutex for enterin unloading quay

HANDLE semADT, \*semVessADT;

HANDLE mutexConsole;//shared mutex between processes

HANDLE mutexBarrier;//mutexBarrier to managed entering to Barrier

HANDLE\* Cranes;//pointer to HANDELE for each Crane thread

unsigned int\* CranesRand;//pointer for array of data holders for Crand random weight generator

int\* CranesID;//pointer to int ,after dynamicly allocation will hold threads-ID array of Cranes

int\* CranesArr;//

// Crain size holds number of Cranes,freeCrains to managed available Crane for unloading,activeCrane indicate if Craine is active

int CraneSize,freeCranes,activeCrane;

HANDLE\* Barrier;//pointer to HANDLE for Array of Scheduling Constraints Semaphores - one for Each Barrier space

int BarrierSize;

int QueueIN, QueueOut;//holds location to entring and exiting from Barrier

int IsFree,tempCountVess;//IsFree to nitify when unloading quay is free,tempCountVess to control QueueIN increasing

SYSTEMTIME lt; //time stracture

char AnsReq[BUFFER\_SIZE];//on test

char buffer[BUFFER\_SIZE]; //to holds the information from the pipe

unsigned int mainRand;

//Thread function for each Vessel

DWORD WINAPI Vessel(PVOID);

//Thread function for each Crain

DWORD WINAPI Crane(PVOID);

//to Initialise and clean global data (mainly for creating the Semaphore Haldles

//before all Threads start and closing them properly after all Threads finish!)

int initGlobalData();

void cleanupGlobalData();

//function decleration

void enterBarrier(int);

void CrossToHaifa(int);

void exitBarrier();

void enterUnloadingQuay(int);

void exitUnloadingQuay(int);

int IsPrime(int);

int IsDivisor(int ,int);

int GenerateNumber(int,int\*);

int CarinsValidNum(int ,int);

int GemerateWeight(int);

int calcSleepTime();

const char\* getTimeString();

void PrintToConsole(char\*);

int main(int argc,char\* argv[])

{

BOOL success;

int check,vessIdIn;

char stringToPrint[STRING\_SIZE];

VessIN = 0;

activeCrane = 0;

QueueIN = 0;

QueueOut = 0;

IsFree = True;

ReadHandle = GetStdHandle(STD\_INPUT\_HANDLE);

WriteHandle = GetStdHandle(STD\_OUTPUT\_HANDLE);

if ((WriteHandle == INVALID\_HANDLE\_VALUE) ||(ReadHandle == INVALID\_HANDLE\_VALUE))

{

fprintf(stderr, "EilatPort: last error %d <\n", GetLastError());

ExitProcess(1);

}

success = ReadFile(ReadHandle, buffer, BUFFER\_SIZE, &read, NULL);

// now have the child read from the pipe

//while((success = ReadFile(ReadHandle, buffer, BUFFER\_SIZE, &read, NULL)))

//{

//

// if (VesselSize != 0)

// break;

// //success = ReadFile(ReadHandle, buffer, BUFFER\_SIZE, &read, NULL);

//}

// we have to output to stderr as stdout is redirected to the pipe

if (!success)

{

fprintf(stderr, "EilatPort: error reading from pipe\n");

return -1;

}

fprintf(stderr, "[%s] EilatPort: Received request of - %s - Vessels \n", getTimeString(), buffer);

VesselSize = atoi(buffer);

check = IsPrime(VesselSize);

if (check != 0)

{

strcpy(AnsReq, "FALSE");

if (!WriteFile(WriteHandle, AnsReq, BUFFER\_SIZE, &written, NULL))

fprintf(stderr, "EilatPort: Error writing to pipe\n");

ExitProcess(1);

}

CraneSize = CarinsValidNum(VesselSize, 2);

sprintf(stringToPrint, "[%s] EilatPort: CraneSize %d \n", getTimeString(), CraneSize);

PrintToConsole(stringToPrint);

BarrierSize = VesselSize / CraneSize;

sprintf(stringToPrint, "[%s] EilatPort: BarrierSize %d \n", getTimeString(), BarrierSize);

PrintToConsole(stringToPrint);

freeCranes = CraneSize;

VessOut = VesselSize;

strcpy(AnsReq, "TRUE");

// now write amended string to the pipe

if (!WriteFile(WriteHandle, AnsReq, BUFFER\_SIZE, &written, NULL))

fprintf(stderr, "EilatPort: Error writing to pipe\n");

if (initGlobalData() == False)

{

fprintf(stderr, "EilatPort: Error while initiating global data \n");

return -1;

}

//read Vessels from the pipe until nummber VessIn equal to received Vesselsize

while (VessIN <VesselSize)

{

// now have the child read from the pipe

success = ReadFile(ReadHandle, buffer, BUFFER\_SIZE, &read, NULL);

// we have to output to stderr as stdout is redirected to the pipe

if (!success)

{

fprintf(stderr, "EilatPort: error reading from pipe\n");

return -1;

}

sprintf(stringToPrint, "[%s] Vessel %s arrived @ Eilat Port \n", getTimeString(), buffer);

PrintToConsole(stringToPrint);

// creat Vessel threads with VesselId unique grather then 1.

vessIdIn = atoi(buffer);

VesselsID[vessIdIn-1] = vessIdIn;

Vessels[vessIdIn-1] = CreateThread(NULL, 0, Vessel, &VesselsID[vessIdIn-1], 0, &ThreadID);

VessIN++;

}

WaitForMultipleObjects(CraneSize,Cranes,TRUE,INFINITE);

sprintf(stringToPrint, "[%s] Eilat Port: All Crane Threads are done \n", getTimeString());

PrintToConsole(stringToPrint);

WaitForMultipleObjects(VesselSize, Vessels, TRUE, INFINITE);

sprintf(stringToPrint, "[%s] Eilat Port: All Vassel Threads are done \n", getTimeString());

PrintToConsole(stringToPrint);

sprintf(stringToPrint, "[%s] Eilat Port: Exiting... \n", getTimeString());

PrintToConsole(stringToPrint);

cleanupGlobalData();

return 0;

}

//The Thread Function for each Crane, gets the Crane Unique ID (index of the Thread in Array of Threads within the main program)

//This Unique Thread ID (called carneID) is used by the Vessel who attached him to unload Vessel carriege

DWORD WINAPI Crane(PVOID Param)

{

//Get the Unique ID from Param and keep it in a local variable

int CraneID = \*(int\*)Param;

char stringToPrint[STRING\_SIZE];

sprintf(stringToPrint, "[%s] crane - %d has been created \n", getTimeString(), CraneID);

PrintToConsole(stringToPrint);

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

{

WaitForSingleObject(semADT, INFINITE);

activeCrane++;

Sleep(calcSleepTime());

sprintf(stringToPrint, "[%s] vessel %d - unloaded %d Tons at Crane %d\n", getTimeString(), CranesArr[CraneID-1], GemerateWeight(CraneID), CraneID);

PrintToConsole(stringToPrint);

Sleep(calcSleepTime());

activeCrane--;

if (!ReleaseSemaphore(semVessADT[CraneID-1], 1, NULL))

fprintf(stderr, "EilatPort->Crane :: Unexpected error sem.V()\n");

}

WaitForSingleObject(semADT, INFINITE);

sprintf(stringToPrint, "[%s] crane - %d done working \n", getTimeString(), CraneID);

PrintToConsole(stringToPrint);

return 0;

}

//The Thread Function for each Vessel, gets the Vessl Unique ID (index of the Thread in Array of Threads within the main program)

//This Unique Thread ID (called VessID) is used by the state DB to keep track of each Vessel state at any given time!

DWORD WINAPI Vessel(PVOID Param)

{

//Get the Unique ID from Param and keep it in a local variable

int VesselID = \*(int\*)Param;

//srand((unsigned)time(NULL));

Sleep(calcSleepTime());

enterBarrier(VesselID);

Sleep(calcSleepTime());

enterUnloadingQuay(VesselID);

Sleep(calcSleepTime());

exitUnloadingQuay(VesselID);

Sleep(calcSleepTime());

CrossToHaifa(VesselID);

Sleep(calcSleepTime());

VessOut--;

if (VessOut == 0)

{

if (!ReleaseSemaphore(semADT, CraneSize, NULL))

printf("EilatPort.Vessel::Unexpected error mutex.V()\n");

}

return 0;

}

//function to control incoming Vessels,when the incoming Vessel is equal to the number of the Cranes at the Unloading Quay

//and Unloading Quay is free the Barrier is released

//used for syncronization point for the process

void enterBarrier(int VessID)

{

char stringToPrint[STRING\_SIZE];

WaitForSingleObject(mutexBarrier, INFINITE);

tempCountVess++;//how many threads entered so far,initialize to zero when the Barrier[QueueIn] is full

if ((IsFree == True) && (QueueIN == QueueOut) && (tempCountVess == CraneSize))

{

QueueIN++;

tempCountVess = 0;

//mutexBarrier.V() - release mutex

if (!ReleaseMutex(mutexBarrier))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

exitBarrier();

}

else

{

if ((IsFree == True) && (QueueIN > QueueOut))

{

exitBarrier();

if (tempCountVess == CraneSize)

{

QueueIN++;

tempCountVess = 0;

sprintf(stringToPrint, "[%s] Vessel %d - entering Barrier \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

//mutexBarrier.V() - release mutex

if (!ReleaseMutex(mutexBarrier))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

WaitForSingleObject(Barrier[QueueIN-1], INFINITE);

}

else

{

sprintf(stringToPrint, "[%s] Vessel %d - entering Barrier \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

//mutexBarrier.V() - release mutex

if (!ReleaseMutex(mutexBarrier))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

WaitForSingleObject(Barrier[QueueIN], INFINITE);

}

}

else

{

if (tempCountVess == CraneSize)

{

QueueIN++;

tempCountVess = 0;

sprintf(stringToPrint, "[%s] Vessel %d - entering Barrier \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

//mutexBarrier.V() - release mutex

if (!ReleaseMutex(mutexBarrier))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

WaitForSingleObject(Barrier[QueueIN - 1], INFINITE);

}

else

{

sprintf(stringToPrint, "[%s] Vessel %d - entering Barrier \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

//mutexBarrier.V() - release mutex

if (!ReleaseMutex(mutexBarrier))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

WaitForSingleObject(Barrier[QueueIN], INFINITE);

}

}

}

sprintf(stringToPrint, "[%s] Vessel %d - exiting Barrier \n", getTimeString(), VessID);

PrintToConsole(stringToPrint);

}

//reales the Barier when the Barrier is ready for release

void exitBarrier()

{

char stringToPrint[STRING\_SIZE];

sprintf(stringToPrint, "[%s] Barrier Release \n", getTimeString());

PrintToConsole(stringToPrint);

if (!ReleaseSemaphore(Barrier[QueueOut], CraneSize, NULL))

fprintf(stderr, "exitBarrier::Unexpected error Barrier[%d].V()\n", QueueOut);

QueueOut++;

IsFree = False;

}

//CrossToHaifa function for Vessels VessID

void CrossToHaifa(int vessID)

{

char stringToPrint[STRING\_SIZE];

//Access to state DB is done exclusively (protected by mutex)

//mutexPipe.P()

WaitForSingleObject(mutexPipe, INFINITE);

sprintf(AnsReq, "%d", vessID);

sprintf(stringToPrint, "[%s] Vessel %d entering Canal: Red Sea ==> Med. Sea \n", getTimeString(), vessID);

PrintToConsole(stringToPrint);

Sleep(calcSleepTime());

/\* the parent now wants to write to the pipe \*/

if (!WriteFile(WriteHandle, AnsReq, BUFFER\_SIZE, &written, NULL))

fprintf(stderr, "Eilat ::Error writing to pipe\n");

//mutexPipe.V() - release mutex

if (!ReleaseMutex(mutexPipe))

printf("Eilat :: Unexpected error mutexPipe.V()\n");

}

void enterUnloadingQuay(int VessID)

{

int availCrain;

char stringToPrint[STRING\_SIZE];

WaitForSingleObject(mutex, INFINITE);

for (availCrain = 0; availCrain < CraneSize; availCrain++)

{

if (CranesArr[availCrain] == -1)

{

sprintf(stringToPrint, "[%s] Vessel %d -settles down at Crane %d \n", getTimeString(), VessID, availCrain+1);

PrintToConsole(stringToPrint);

CranesArr[availCrain] = VessID;

freeCranes--;

break;

}

}

// mutex.V() - release mutex

if (!ReleaseMutex(mutex))

printf("UnloadingQuay::Unexpected error mutex.V()\n");

if (freeCranes == 0)//if there is no more free crane signal crane threads

{

if (!ReleaseSemaphore(semADT,CraneSize,NULL))

printf("UnloadingQuay::Unexpected error mutex.V()\n");

}

WaitForSingleObject(semVessADT[availCrain], INFINITE);

}

void exitUnloadingQuay(int VessID)

{

int availCrain;

char stringToPrint[STRING\_SIZE];

WaitForSingleObject(mutex, INFINITE);

for (availCrain = 0; availCrain < CraneSize; availCrain++)

{

if (CranesArr[availCrain] == VessID)

{

sprintf(stringToPrint, "[%s] Vessel %d -left Crane %d \n", getTimeString(), VessID, availCrain+1);

PrintToConsole(stringToPrint);

CranesArr[availCrain] = -1;

freeCranes++;

break;

}

}

// mutex.V() - release mutex

if (!ReleaseMutex(mutex))

printf("UnloadingQuay::Unexpected error mutex.V()\n");

if (freeCranes == CraneSize)

{

IsFree = True;

if ((VessIN == VesselSize) && (QueueIN>QueueOut))

{

exitBarrier();

}

}

}

//checks if num is a Prime number

int IsPrime(int num)

{

int i;

for (i = 2; i\*i <= num; i++) {

if (num % i == 0)

return 0;

}

return 1;

}

//checks if num is a divisor of the limit number

int IsDivisor(int num, int limit)

{

if (limit % num == 0 && limit != num)

return 1;

return 0;

}

//secure random number generator

int GenerateNumber(int limit,int\* randNum)

{

errno\_t err;

unsigned int tempRand;

err = rand\_s(randNum);

if (err != 0)

{

printf\_s("The rand\_s function failed!\n");

}

tempRand = ((unsigned int)\*randNum) % limit + 1;

return tempRand;

}

//generate valid CraneSize

int CarinsValidNum(int limit, int min)

{

int temp = GenerateNumber(limit,&mainRand);

while (!IsDivisor(temp, limit) || temp < min)

{

temp = GenerateNumber(limit,&mainRand);

}

return temp;

}

// Generate weight for Crain thread when Vessel unload his carriege at the Unloading Quay

int GemerateWeight(int CranesID)

{

int randWeight = 0;

while (randWeight < 5)

{

randWeight = GenerateNumber(MAX\_WEIGHT,&CranesRand[CranesID-1]);

}

return randWeight;

}

//generic function to randomise a Sleep time between MIN\_SLEEP\_TIME and MAX\_SLEEP\_TIME msec

int calcSleepTime()

{

int randSleep = 0;

while (randSleep < MIN\_SLEEP\_TIME)

{

randSleep = GenerateNumber(MAX\_SLEEP\_TIME,&mainRand);

}

return randSleep;

}

//safe console printing

void PrintToConsole(char \*string)

{

WaitForSingleObject(mutexConsole, INFINITE);

fprintf(stderr, "%s", string);

if (!ReleaseMutex(mutexConsole))

printf("PrintToConsole::Unexpected error mutex.V()\n");

}

//return const string of current locl time

const char\* getTimeString()

{

GetLocalTime(&lt);

static char currentLocalTime[20];

sprintf(currentLocalTime, "%02d:%02d:%02d", lt.wHour, lt.wMinute, lt.wSecond);

return currentLocalTime;

}

//Initialise global Semaphores

//If all Successful - return True, otherwise (if problem arises) return False

//This is invoked before all Vessels Threads start running

int initGlobalData()

{

int i;

mutex = CreateMutex(NULL, FALSE, NULL);

if (mutex == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating mutex\n");

return False;

}

mutexPipe = CreateMutex(NULL, FALSE, NULL);

if (mutexPipe == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating mutexPipe\n");

return False;

}

mutexConsole = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, TEXT("ConsoleMutex"));

if (mutexConsole == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating mutexConsole\n");

return False;

}

mutexBarrier = CreateMutex(NULL, FALSE, NULL);

if (mutexBarrier == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating nutexBarrier\n");

return False;

}

semADT = CreateSemaphore(NULL,0,CraneSize, NULL);

if (semADT == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating semADT\n");

return False;

}

semVessADT = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, (CraneSize \* sizeof(HANDLE)));

if (semVessADT == NULL)

{

fprintf(stderr, "initGlobalData:: Error creating semADT\n");

return False;

}

Vessels = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, (VesselSize\*sizeof(HANDLE)));

if (Vessels == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating Vessels\n");

return False;

}

Barrier = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, (BarrierSize \* sizeof(HANDLE)));

if (Barrier == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating Barrier\n");

return False;

}

Cranes = HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, (CraneSize \* sizeof(HANDLE)));

if (Cranes == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating Cranes\n");

return False;

}

CranesID = malloc(CraneSize \* sizeof(int));

if (CranesID == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating CranesID\n");

return False;

}

CranesRand = malloc(CraneSize \* sizeof(unsigned int));

if (CranesRand == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating CranesRand\n");

return False;

}

VesselsID = malloc(VesselSize \* sizeof(int));

if (VesselsID == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating VesselsID\n");

return False;

}

CranesArr = malloc(CraneSize \* sizeof(int));

if (CranesArr == NULL)

{

fprintf(stderr, "initGlobalData:: Error allocating CranesArr\n");

return False;

}

for (i = 0; i < BarrierSize; i++)

{

Barrier[i] = CreateSemaphore(NULL, 0, CraneSize, NULL);

if (Barrier[i] == NULL)

{

return False;

}

}

for (i = 0; i < CraneSize; i++)

{

CranesID[i] = i+1;

CranesArr[i] = -1;

CranesRand[i] = 0;

Cranes[i] = CreateThread(NULL, 0, Crane, &CranesID[i], 0, &ThreadID);

semVessADT[i] = CreateSemaphore(NULL, 0, 1, NULL);

}

return True;

}

//Close all global semaphore handlers and free dynamic allocations - after all Vessels Threads finish.

void cleanupGlobalData()

{

int i;

CloseHandle(mutexPipe);

CloseHandle(semADT);

CloseHandle(mutex);

CloseHandle(mutexBarrier);

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

{

CloseHandle(Vessels[i]);

}

for (i = 0; i < BarrierSize; i++)

{

CloseHandle(Barrier[i]);

}

for (i = 0; i < CraneSize; i++)

{

CloseHandle(Cranes[i]);

CloseHandle(semVessADT[i]);

}

free(CranesID);

free(CranesArr);

free(VesselsID);

free(CranesRand);

HeapFree(GetProcessHeap(), HEAP\_NO\_SERIALIZE, Barrier);

HeapFree(GetProcessHeap(), HEAP\_NO\_SERIALIZE, Vessels);

HeapFree(GetProcessHeap(), HEAP\_NO\_SERIALIZE, Cranes);

HeapFree(GetProcessHeap(), HEAP\_NO\_SERIALIZE, semVessADT);

}

*Running examples*

*4 vessels*

[](https://player.vimeo.com/video/443175035?app_id=122963)

*8 vessels*

*[](https://player.vimeo.com/video/443176911?app_id=122963)*

*16 vessels*

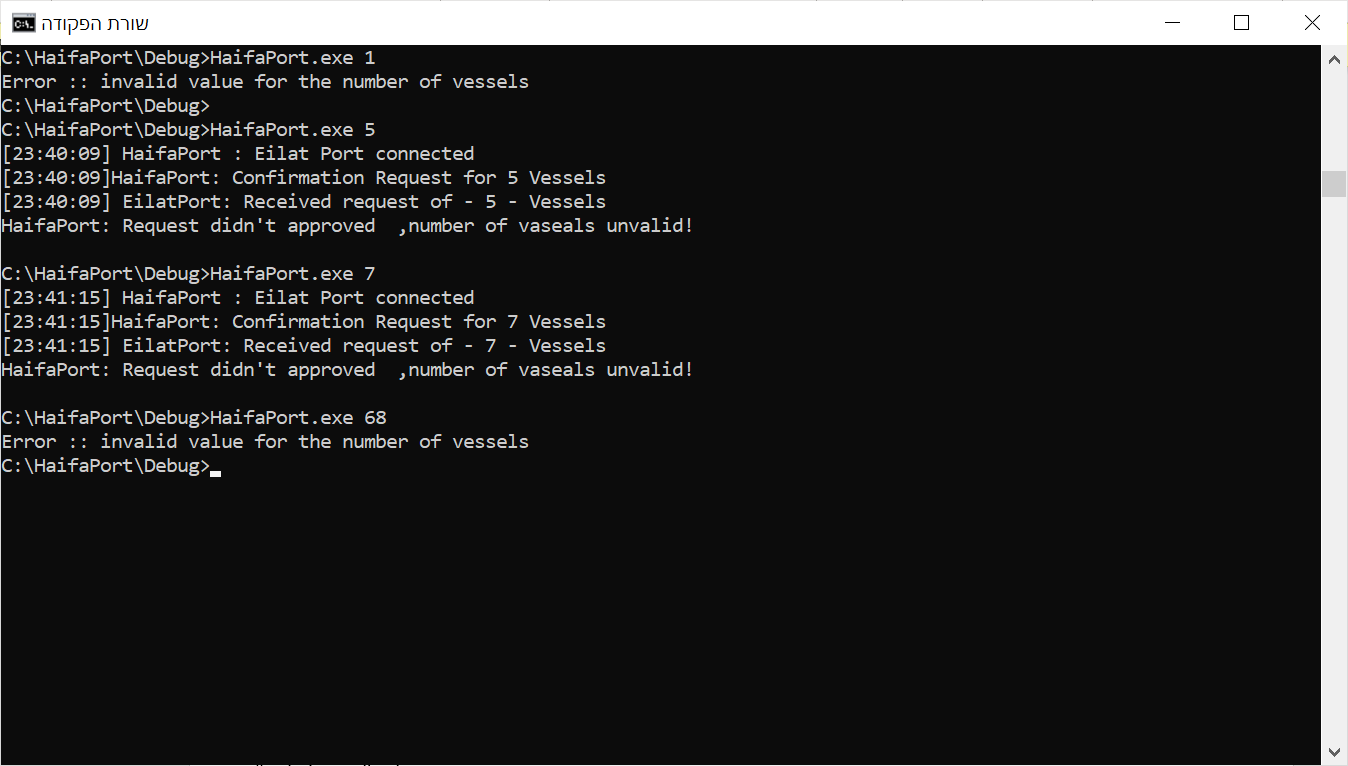
[](https://player.vimeo.com/video/443178492?app_id=122963)

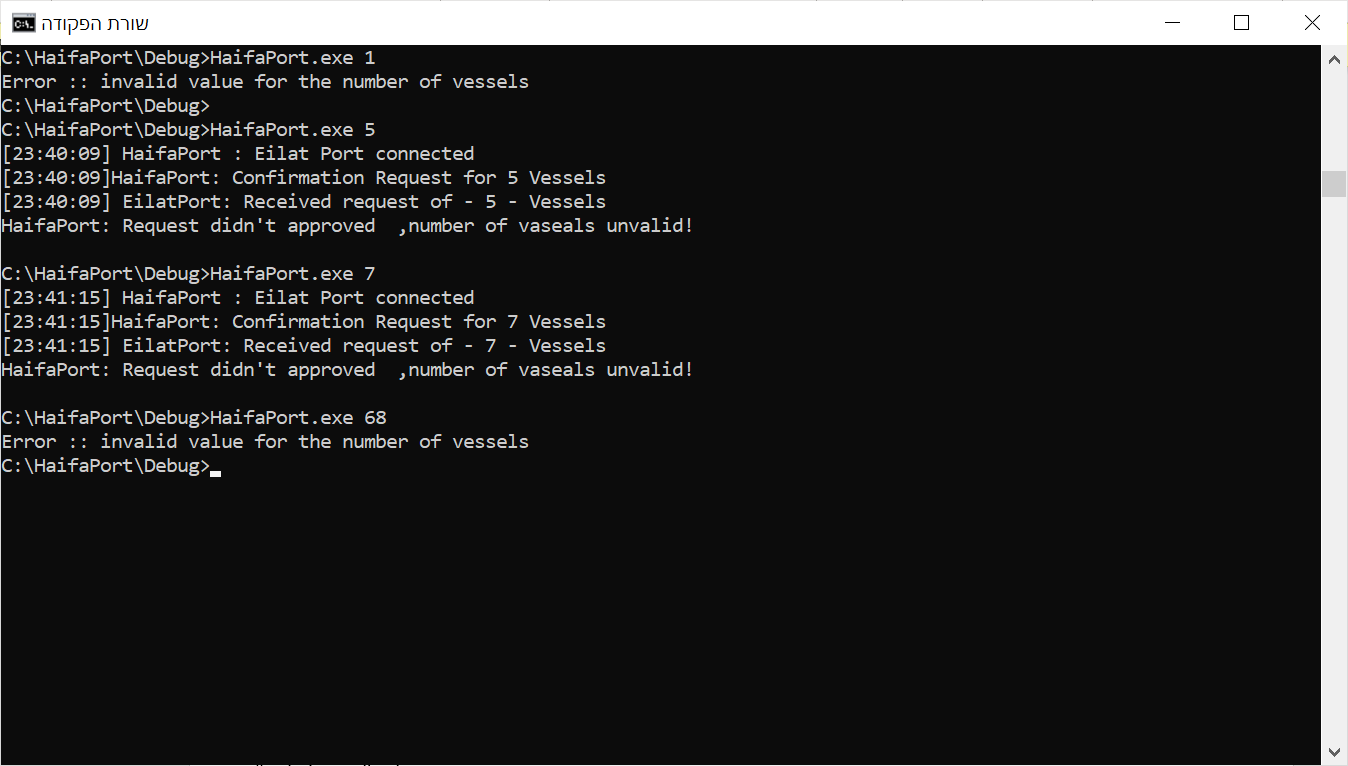
*35 vessels*

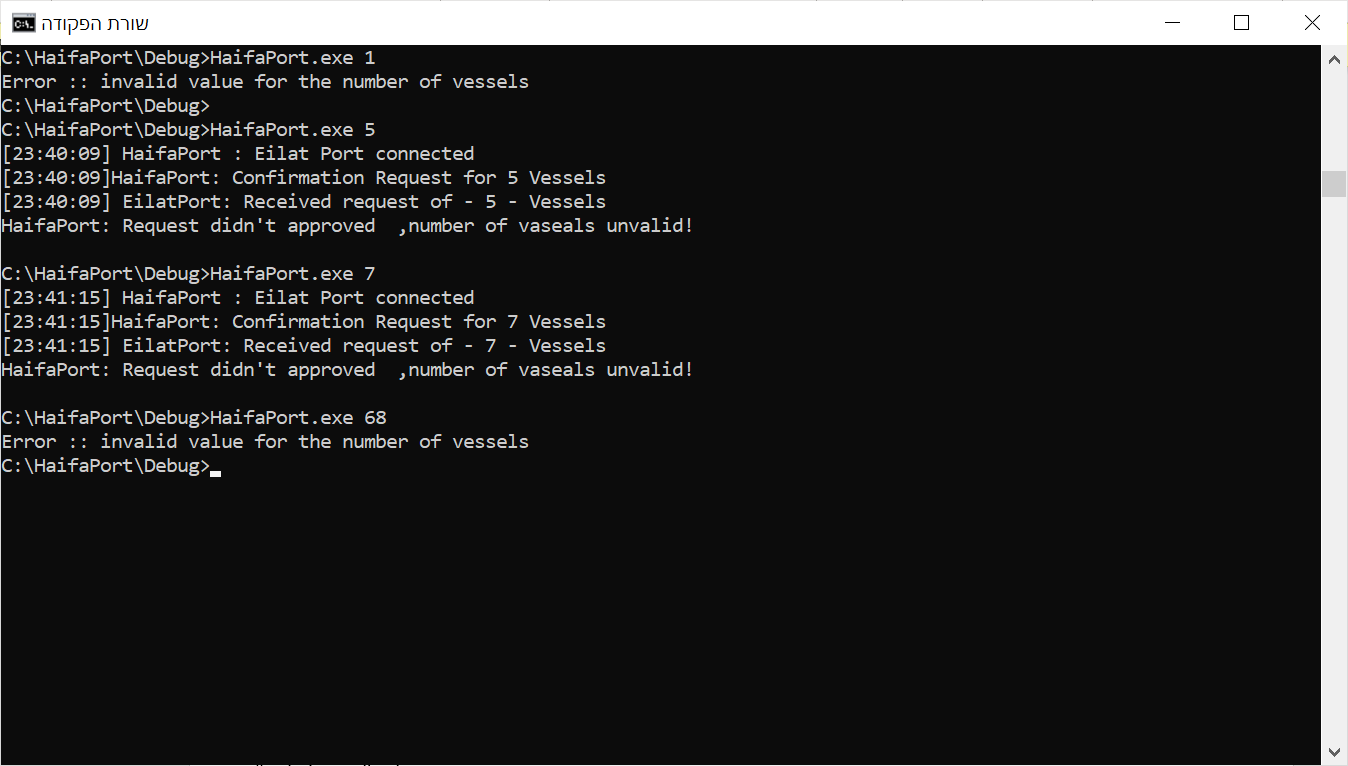
[](https://player.vimeo.com/video/443180791?app_id=122963)

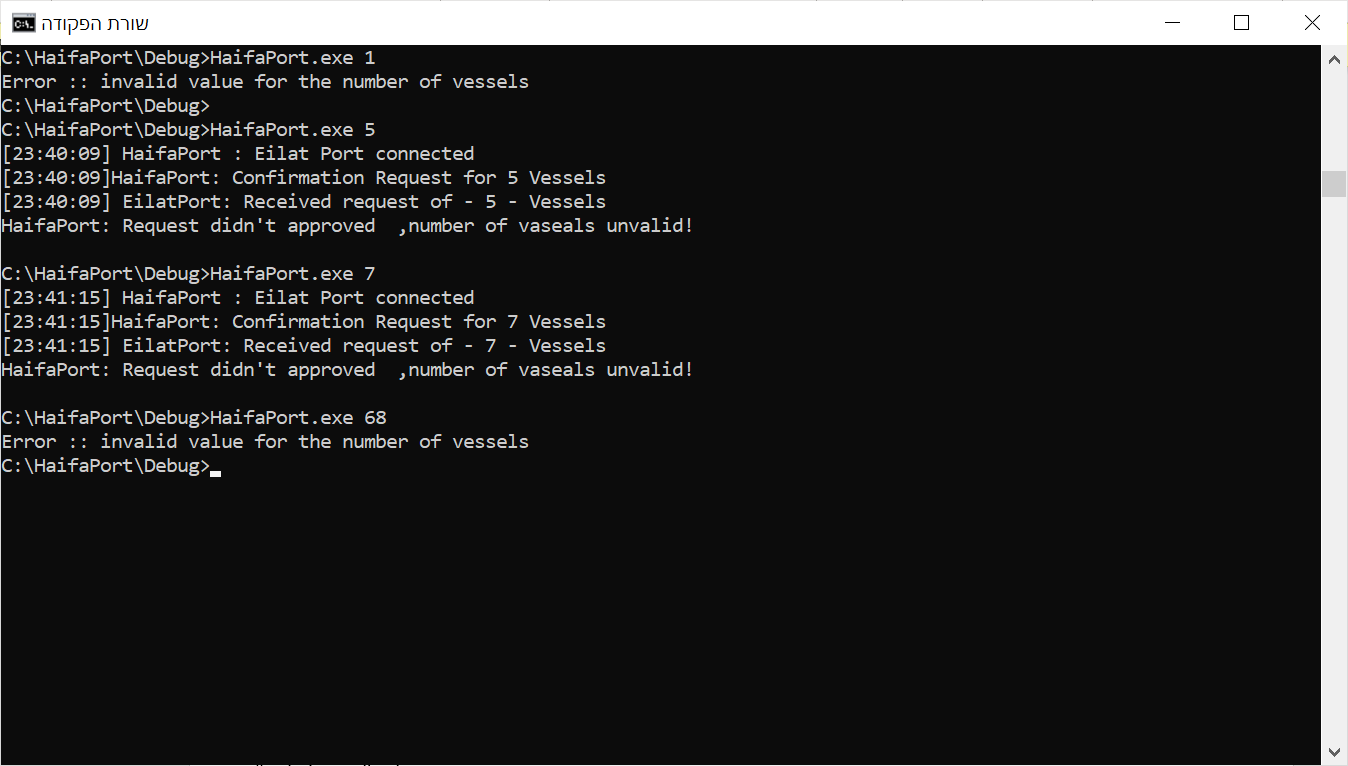
*50 vessels*

[](https://player.vimeo.com/video/443181464?app_id=122963)









|  |  |  |
| --- | --- | --- |
|  |  |  |
| “The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.” ― Edsger W. Dijkstra |
|  |