# CSCE 313-200: Computer Systems

Homework #1 Part 1 (25 pts)

Due date: 1/23/25

## 1. Purpose

Understand the Visual Studio environment, creation of projects, simple process debugging, search algorithms, and basic inter-process communication.

## 2. Problem Description

Your goal is to implement a number of multi-threaded search algorithms on weighted graphs being held by another process in the system. This conveniently maps to the following problem that might be easier to understand. Assume that your job is to navigate a space rover out of a cave located on some remote planet. Each cave consists of a maze of (mostly) dark rooms interconnected by tunnels, all of which can be represented by a giant undirected graph (i.e., each room is a node, each tunnel between rooms is an edge).

Since the rover is slow, it cannot search for the exit directly. However, it can unleash *N* search flybots to explore the cave. Due to their small size, these robots are relatively dumb, which means that they can neither remember the rooms they have been to nor coordinate with each other to avoid covering the same room multiple times. Each flybot is equipped with a wireless link that allows it to receive navigation directions from the rover (i.e., which room to explore next). The result of each visit is the list of neighboring rooms and the amount of light visible to the flybot in each of them, which are transmitted back to the rover to aid the search. The high-level objective is to concurrently control the robots so as to find the unique exit in the shortest amount of time.

The topology of the cave is unknown a-priori and must be explored in real-time. The rover contains a module called *Command Center* (CC), which relays directives from your software to the robots and forwards their responses back to you. See Figure 1 for an illustration. The CC process and its standardized control directives are provided as part of this homework. During execution, you must first launch the CC and then communicate with it for all subsequent navigation of the cave through a message-based channel (or multiple channels).

It should be noted that responses from flybots are not instantaneous since there is some inherent delay needed to move from one room to another. This results not only in large, but also randomly fluctuating, response delays. This puts severe constraints on how much exploration can be done with a single robot. However, with multi-threading you should be able to successfully escape caves with $10^7$ nodes.

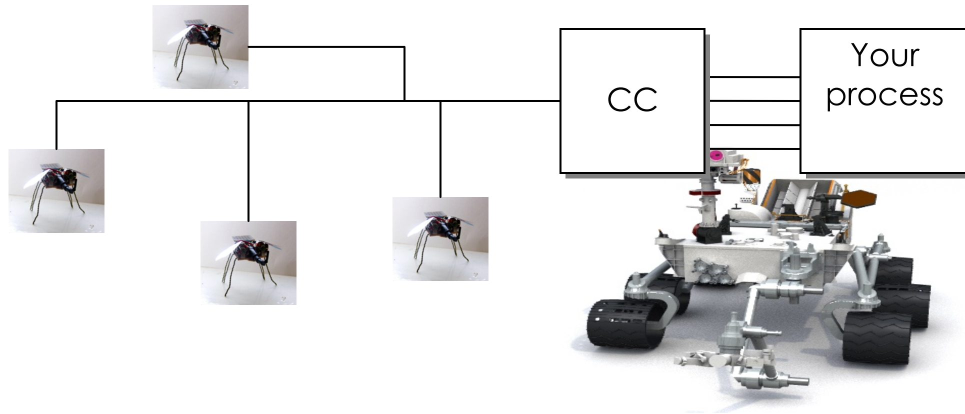


**Figure 1. Operation of the CC with 4 flybots.**

During the initial connection to the CC, you must specify a particular planet $P$ and cave $C$ within that planet, where $1 \leq P \leq 7$ determines how large the cave is (size = $10^P$ rooms) and $C \geq 0$ specifies which random instance of that cave you will attempt to navigate. For example, tuple (3, 7) refers to the 7th random instance of some cave with 1,000 rooms. This notation allows you to test your code with small caves to start with (i.e., $P$ = 1 or 2) and then transition to larger ones (i.e., $P$ = 7) as your program becomes more efficient. In addition, this model allows you to replay old scenarios (i.e., by specifying the same $C$), while having a virtually unlimited supply of new caves to experiment with (i.e., by varying $C$).

## 2.1. Code (25 pts)

The program must accept two command-line arguments that identify the planet and cave to be explored:

```
caveSearch.exe 6 44
```

where this example means planet 6, cave 44. Your code should check if the number of input parameters is two and otherwise print usage info.

You must be able to connect to the CC, print its response, then open a new channel to the first robot, issue another connect, and print the initial room ID from that response. Then, your program must cleanly shut down the robot, then the CC, and finally terminate after the CC process exits. Your printouts should be of this form:

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 4, cave 4, robots 1...
*** CC: creating graph (1st pass)
*** CC: creating graph (2nd pass)
*** CC: building a map
*** CC: propagating light sources
*** CC: finished initialization
CC says: status = 1, msg = 'opened (planet 4, cave 4) with 1 robot(s)'
Connecting to robot 0...
Robot says: status = 1, msg = 'connected'
Current position: room 8DCD0AE66BDA38D9, light intensity 0.00
-------------------------
Disconnecting robot 0...
```

```
Disconnecting CC...
Waiting for CC.exe to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 0.00 sec, user time 0.01 sec
Execution time 1.77 seconds
```

Note that every line that starts with *** comes from CC.exe, while everything else is printed by your program. Also observe that the execution time must be displayed only *after* CC.exe has quit.

Efficient coding and well-structured programming is expected. You should not copy-paste the same function (with minor changes) over and over again, write poorly designed or convoluted code, or allow buffer overflows, access violations, debug-assertion failures, heap corruption, synchronization bugs, memory leaks, or conditions that lead to a crash. *Also make sure to check for errors in every API you call.*

## 2.2. Process Creation

The CC process should be downloaded from the course webpage and placed into the same folder as your cpp/h files. From there, creating a process is rather simple:

```
PROCESS_INFORMATION pi;
STARTUPINFO s;

GetStartupInfo (&s);

char path [] = "CC.exe";
if (CreateProcess (path, NULL, NULL, NULL,
        false, 0, NULL, NULL, &s, &pi) == FALSE)
{
        printf ("Error %d starting %s\n", GetLastError(), path);
        exit(-1);
}
```

On return, structure `pi` contains two important fields – the *handle* of the created process and its *process ID* (PID). Using the former, you will need to pause at the end of `main()` until the CC process terminates (see `WaitForSingleObject`). Using the latter, you will open a named pipe and communicate with the CC as described next.

## 2.3. Named Pipes

This homework explores an inter-process communication primitive called *named pipe*, which is a bidirectional, lossless, synchronous channel between two processes. The first task in opening a pipe is to construct its name. To prevent multiple instances of the CC from interfering with each other on the same host, each pipe name must carry the hex PID of the CC. For example, `\\.\pipe\CC-17A5` is the name of a pipe that can be opened to a CC process with PID 17A5. You can use `sprintf` or STL string functions to create this name. Note that the PID comes from `CreateProcess` in Section 2.2.

There are two general steps needed to open a pipe once its name is known:

```
// wait for the CC to initialize the pipe from its end
while (WaitNamedPipe (pipename, INFINITE) == FALSE)
        Sleep (100);                  // pause for 100 ms
```

```
// now open the pipe
HANDLE CC = CreateFile (pipename,  GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

If creation is successful, the pipe can be read from and written to using `ReadFile` and `WriteFile`. To close the pipe or any other open handle (e.g., mutex, semaphore), use `CloseHandle`.

## 2.4. CC Message Format

Make sure to pack all structs below to 1 byte, i.e., use `#pragma pack(push,1)` before declaring them and `#pragma pack(pop)` afterwards. After establishing a pipe, you can send messages to the CC, each consisting of the following 7 bytes:

```
class CommandCC {
public:
       uchar                 command:2;     // lower 2 bits
       uchar                 planet:6;      // remaining 6 bits
       DWORD                 cave;          // which cave
       ushort                robots;        // how many robots
};
```

Here, `command` can be either CONNECT = 0 or DISCONNECT = 1. In the former case, the next two parameters (planet, cave) must be those provided by the user at the command line, while the last one (i.e., number of robots) should be hardcoded to 1. The CONNECT command elicits the following response:

```
class ResponseCC {
public:
       DWORD                 status;
       char                  msg [64];
};
```

The status code can be either FAILURE = 0 or SUCCESS = 1, while `msg` is a NULL-terminated char buffer providing a text response to the command. You can directly use `printf` on the message.

The DISCONNECT command, where triple (planet, cave, robots) can be arbitrary, requests that the CC shut down, which it does after waiting for all flybot threads to quit. There is no response to DISCONNECT.

## 2.5. Robot Message Format

Once the CC is aware of the number of desired robots *N*, it will start this many internal threads and create a new pipe for each of them. For a given CC pipe `\\.\pipe\CC-17A5`, robot pipes will have names `\\.\pipe\CC-17A5-robot-0`, `\\.\pipe\CC-17A5-robot-1`, and so on up to *N*–1, where *N* was requested in the initial CONNECT message. Note that numbers 0, 1, 2, …, *N*–1 must be written in *hex*. Use `_itoa` or `sprintf` to accomplish this conversion. For example:

```
char robotPipeName [1024];
int k = 5;            // 5th robot
sprintf (robotPipeName, "\\\\.\\pipe\\CC-%X-robot-%X", PID, k);
```

Next, you will need to sleep-spin on `WaitNamedPipe` until these pipes become available and afterwards issue `CreateFile` on each of them as explained in Section 2.3.

Robot requests have the following format:

```
class CommandRobot {
public:
       DWORD                 command;
```

```
	uint64			room;		// unsigned __int64
};
```

where the command can be CONNECT = 0, DISCONNECT = 1, or MOVE = 2. For the first two commands, the room parameter is ignored (although it still must be provided in the message). For the last command, the room specifies where the robot should fly to.

Robot responses start with a header that follows this structure:

```
class ResponseRobot {
public:
	DWORD			status;
	char			msg [64];
};
```

The `status` and `msg` fields are identical to those in CC responses. If the command is successful, the `ResponseRobot` header is followed by a list of neighboring room IDs (`uint64`) and light intensities (`float`). Upon failure, the message stops at the header. Starting with Part 2, the total length of each response will be determined using `PeekNamedPipe`.

After the entire message is obtained from the OS via `ReadFile`, you can use a pointer of type `NodeTuple64` to read the list of neighbors:

```
class NodeTuple64 {
public:
	uint64			node;
	float			intensity;
};
```

The CONNECT command returns only one room in the response, which is where the rover is located. This room is the starting point of your search in Part 2.

The DISCONNECT command shuts down the robot and does not elicit any response. It should be noted that the proper shutdown sequence is to first stop all robots, close their *N* pipes, then stop the CC, close its pipe, and finally wait for the CC process to quit.

Avoid hardwiring constants; instead, apply `sizeof()` to every class and use the following definitions:

```
#define CONNECT				0
#define DISCONNECT			1
#define MOVE				2

#define FAILURE				0
#define SUCCESS				1
```

## 2.6. Debug Printouts

One useful technique is to print raw buffers byte-by-byte when you are unsure what is going on. For example, suppose `bufSize` bytes have just been received from the pipe into `someBuffer`. Then, you can dump the entire buffer to screen using a simple loop:

```
char *someBuffer;
for (int i = 0; i < bufSize; i++)
	printf ("%X ", someBuffer [i]);
```

For more information on `printf`, see:

[https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=vs-2019](https://docs.microsoft.com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=vs-2019)

## 2.7. Traces

Sample run:

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 1, cave 1, robots 1...
*** CC: creating graph (1st pass)
*** CC: creating graph (2nd pass)
*** CC: building a map
*** CC: propagating light sources
*** CC: finished initialization
CC says: status = 1, msg = 'opened (planet 1, cave 1) with 1 robot(s)'
Connecting to robot 0...
Robot says: status = 1, msg = 'connected'
Current position: room E8825E744AF37DEB, light intensity 2.85
-------------------------
Disconnecting robot 0...
Disconnecting CC...
Waiting for CC.exe to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 0.00 sec, user time 0.07 sec
Execution time 1.12 seconds
```

A few more initial positions:

```
planet 5, cave 15: room 2346CD5D02A2D914, light intensity 0.00
planet 6, cave 1300: room 1971C9F35798C4B2, light intensity 0.00
planet 7, cave 320: room 72E191306EAD0A94, light intensity 0.00
```

The next example shows what happens with an invalid planet. Your program must not continue when it encounters errors, but it may terminate without cleanup in fatal cases. For CC errors to be seen, make sure to *skip validity checks of the input arguments and send them as-is to the CC*.

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 50, cave 1, robots 1...
CC says: status = 0, msg = 'connect error: only planets 1-7 are supported'
Connection error, quitting...
*** CC: error 109 reading the pipe, exiting
```

As shown above, an unclean shutdown forces the CC to throw an error shown in bold (since the kernel closes the pipe after your process quits), which is normal.

# 313 Homework 1 Grade Sheet (Part 1)

Name: ____________________

| **Function** | **Points** | **Break down** | **Item** | **Points** |
|---|---|---|---|---|
| **Basic code structure** | 13 | 2 | Create CC process | |
| | | 1 | Correct CC pipename | |
| | | 1 | Correct robot pipename | |
| | | 1 | Proper connect to CC | |
| | | 1 | Proper connect to robot | |
| | | 1 | Proper read of CC response | |
| | | 2 | Proper read of robot response | |
| | | 1 | Proper robot disconnect | |
| | | 1 | Proper CC disconnect | |
| | | 2 | Wait for CC.exe to quit | |
| **Functionality** | 6 | 2 | Correct initial room shown | |
| | | 2 | Correct intensity shown | |
| | | 2 | Handles CC errors (e.g., invalid planet) | |
| **Printouts** | 6 | 2 | CC response msg | |
| | | 2 | Robot response msg | |
| | | 2 | Execution time | |
| **Misc*** | | | | |

Total points: _________________