– Lab2 –

System Programming in Windows

- how to get a planet to spin

1 Introduction

The purpose of this lab is to get you to try to use the system services that are common in the operating system (OS), and to get a feeling for the pseudo-parallelism, mutual exclusion, and client-server concept. The lab will use the Windows operating system.

The difference in functionality between different OS is usually not so great, but the way that you are using them may be quite different. (Anyone who knows a bit about system calls of UNIX will probably be confused in the beginning.)

We hope that this lab will give you a better understanding of how services in the next lab is intended to work, and give concrete examples of the material reviewed in the lectures. The functionalities that will be addressed in this lab are:

- Creation of processes and threads
- Semi-parallelism
- Communication between processes (IPC)
- Shared resources

Time-frame

You obviously work in your own pace, but it is recommended that you complete this lab duing the third lab session (i.e. "Planetlab II").

Note: Never count on completing the labs during the lab hours!

General

Always check the course website, where you may find new tips, corrections, and explanations of common problems.

2 Assignment

In this lab you will create a simulation environment of a planetary system. This system should be implemented as client-server model, where the server should offer the following services to its clients:

createPlanet(name, mass, position, velocity, life)

where *name* is the name of the plane to be created, the *mass* is the planet mass, *position* of its starting position, *velocity* is the planet's original speed expressed as a vector, and *life* is the planet's life expressed in units of time. (To not make things too complex, we will work in 2-D, thus becoming the "x and y" coordinate of a planet's position and its velocity.)

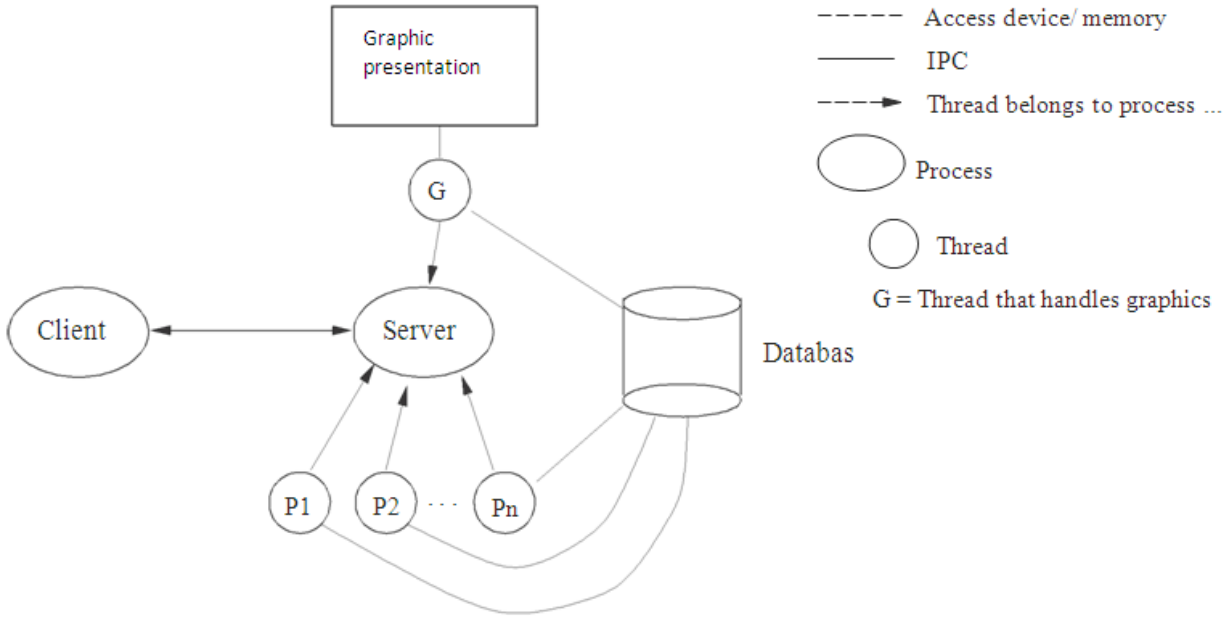

Figure 1: The link between the various components in the lab

The server must have a database (linked list) with active planets (life> 0), and the server should create a thread for each planet. The task of this thread is to calculate the planet's next position by consulting the database server. (The planet's next position depends on its velocity and the other planets positions and masses.) After the planet has calculated its new position, the life of the planet is decremented by one. Finally, the thread updates the database with the new position, life, etc. Figure 1 shows how the different parts of the lab are supposed to be interconnected.

There is no strict time requirements on each planet / thread working completely in sync with the others. In other words, there is no need for synchronization between different planets / threads. When the thread has updated the data, it should sleep for a while.

Below is a derivation of what determines the planets positions. The derivation is just for you information, which is not mandatory to follow completely. You can skip directly to the **Results**.

Derivation:

The physical laws governing the planets positions and velocities are (according to the Newtonian model):

$$F_s = G\frac{m_1 m_2}{r^2} \qquad F = m \cdot a$$

$$v = \int a(t)\, dt \qquad s = \int v(t)\, dt$$

$$G = 6.67259 \times 10^{-11}\, N \cdot m^2/kg^2$$

where s = distance, v = velocity, and a = acceleration.

Example: Figure 2 shows how gravitational forces affect the two planets. Using the above formulas it is possible to derive the acceleration for a plane. If the planet's mass, position and velocity is given, the planet's new position can then be inferred. We perform this for the plane 1 (P1 in Figure 2):

$$F_s = G\frac{m_1 m_2}{r^2} \qquad F_s = m_1 \cdot a_1$$

$$m_1 \cdot a_1 = G\frac{m_1 m_2}{r^2} \Rightarrow a_1 = G\frac{m_2}{r^2} \tag{1}$$

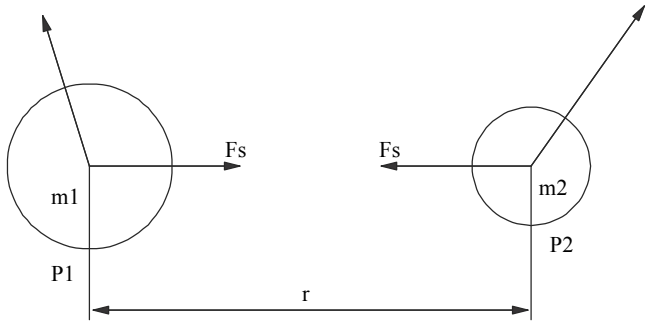

Figure 2: The attraction between two celestial bodies

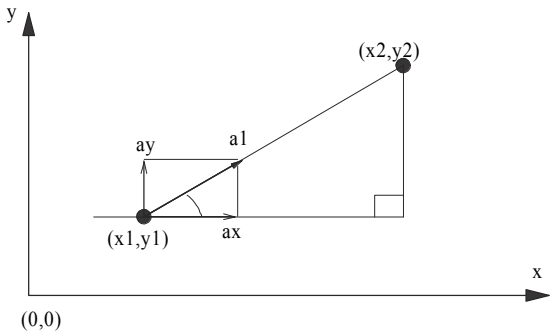

Figure 3: Breakdown of force components Fs

Where r is calculated as follows for the two planets with positions (xl, yl) and (x2, y2) (see Figure 3):

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

These laws are designed in the one-dimensional plane. We must therefore find a way to transfer our two-dimensional problem into a form so that the one-dimensional laws can be used.

This conversion is done by calculating the acceleration for each of the two dimensions, ax and ay in Figure 3

$$cos(\alpha) = \frac{x_2 - x_1}{r} \qquad a_x = a_1\frac{x_2 - x_1}{r} \tag{3}$$

$$sin(\alpha) = \frac{y_2 - y_1}{r} \qquad a_y = a_1\frac{y_2 - y_1}{r} \tag{4}$$

When we have these formulas, we can treat each dimension separately. Positions for each dimension determined by the following formula:

$$s_x = \int v_x(t)\,dt = \int\int a_x(t)\,dt$$

If we approximate the integration by summation, and perform the following calculation for small values in dt, we get:

$$v_x^{new} = v_x^{old} + a_x \cdot dt \quad (5)$$

$$s_x^{new} = s_x^{old} + v_x^{new} \cdot dt \quad (6)$$

Where s_x^{new} is the planet's new position, and s_x^{old} is its old position. The same should be performed for the y-axis.

This is all we need for this lab! Remember that the greater the value of dt is chosen the larger error will be involved in the calculation. (The lab does not require an rigorous precision)

Results:

Below is the pseudo-code to calculate a planet's position when there are several other planets in the vicinity, i.e. this is the code of the planet thread:

1. $a_{tot_x} = 0$, $a_{tot_y} = 0$ (i.e. the total acceleration in each dimension)
2. For each plane (γ) in the database except their own (δ):

   (a) Determine the acceleration contribution from γ to δ, i.e. calculate Equation(1). (Hint: Equation (2) is used to solve Equation (1).)

   (b) Calculate the acceleration for each dimension, as shown in Equation (3) and (4)

   (c) Add the results from above to a_{tot_x} and a_{tot_y} respevtively.
3. From the above steps we get a total contribution of the acceleration in x-and y-dimensions. For each dimension, the planet's next position is calculated using equation (6), and you need Equation (5) to solve it. (To calculate this, you need to know the current position and velocity.)
4. Let the thread sleep for a while. Appropriate time is 10ms. (NOTE: There are two different functions to allow a thread "sleep": *sleep* and *Sleep*. *sleep* sleeps in the given number of seconds, *Sleep* Sleeps in the specified number of milliseconds. Use ***Sleep***!

   Go to Step 1 and repeat the whole procedure forever.

There are two reasons for a planet to be removed from the system:

- planet's lifetime expires (i.e. life = 0)
- Planet's location is outside the bounding box. Appropriate bounding box has size 800 x 600

When a planet is removed from the system, a message should be sent to the client that created the planet, along with the information of why the planet was removed. The client will then present this information appropriately. (Hint: The mailslots of clients must have unique names. In order to create unique client name, the function *GetCurrentProcessId* or *GetCurrentThreadId* are suitable.)

The server must be so general that it can deal with multiple clients simultaneously. The server also has the task of printing out the planets' positions in one window with the size '*aaa * bbb'*. To perform this, we will directly provide a number of functions (you will learn more about these in lab 3). Moreover, each client can have multiple active planets started simultaneously with the server. This means that each client need to have more than one thread. (Hint: More than two threads is usually unnecessary.)

If you want, you are of course encouraged to implement your own graphical function. For this lab, it is sufficient to represent a planet as a pixel in the presentation window. Ideally, the

different planets have different colors when they are printed out. (It helps if the planets leave "tracks" for themselves, so that we can trace the movement of the planets. This is the way we propose for the presentation, but there is nothing to prevent other solutions.)

3 Lab Environment

Using windows

The lab is made in Windows using Visual Studio.

The client should be implemented as a **Console application** and is text based, while the server uses a graphical window. The package on the webpage contains a compilable source that you should start with.

First, try to run the given code and figure out how it works. (Please first run server and then client.)

Then modify the code to create your solution. You will notice that most of the code can be maintained. A normal solution requires that you add approximately 200 lines of code.

Note that the client and the server files in the lab package must be in different Visual Studio solutions (i.e. will be different *exe-files* in the end).

To execute the files, you must include your wrappers from lab 1.

Note: In case the source files do not work, you probably have faults in your wrappers. (Be sure the given source files work correctly before you start your own solution.)

Using Linux

This lab should be implemented as a terminal application, thus you have no need for using an IDE. The client should be created as a single .cpp file, you can find the main function on the website.

Since we will use the graphical library SFML in this lab, you will have to install it to you Linux system. If you use a debian based system such as Ubuntu, you type the following command to install:

```
sudo apt-get install libsfml-dev
```

When you have installed SFML you can compile the code with the following command:

```
g++ main.cpp -o sfml-app -lsfmlgraphics -lsfml-window -lsfml-system
```

You can then run your program using the following command:

```
./sfml-app
```

Study the file, and see what it does, it contains the basic functionality you will need in order to create a planet system. When you feel ready, insert your wrappers as libraries to the main files and try compiling again, now with the additional –lpthread flag. When this is done, start to modify the code to create your solution..

Note that the client and the server files should be maintained separately, the will be run as different binaries at the end.

To execute the files, you must include your wrappers from lab 1.

Note: In case the source files do not work, you probably have faults in your wrappers. (Be sure the given source files work correctly before you start your own solution.)

4 Debugging

Using Windows

In this lab you may not be able to use the debugger to look for errors, because the support for debugging of multithreaded applications is limited. Instead, we recommend you to use print statements to trace the execution.

For graphical applications (i.e. the server side), *printf* does not work. Alternatively, you can use *MessageBox* function to print. One of the benefits of this feature is that it stops the thread that calls the function until the message window closes.

MessageBox function takes one parameter including the window title and a string to be displayed in the window. In order to create strings with values of variables, you can use function *sprintf*.

Example

```
int i = 267;
char str[64];

sprintf(str, "Värdet på i är %d", i)
```

Using Linux

In this lab you may not be able to use an IDE to debug your solutions, but since you are using Linux, it is good getting used to.

To debug .c files without an IDE, you can use printf("%format", value) to print out the value you want a closer look at. Sometimes, the IO buffer doesn't print straight away and therefore it may be a good idea to add fflush(stdout) after a printf.

5 Testing

It can be difficult to know what are the appropriate test values for this lab. Therefore, we provide an example that includes one stationary planet with another planet in a circular orbit around. Use *dt* = 10.

	m	x	y	v_x	v_y
P_1	10^8	300	300	0	0
P_2	1000	200	300	0	0.008

If your program is working properly, the planet P2 circulates around P1. P1 will almost stand still. After this test, it is free to create your own planets with any parameters.

If you have big problems with understanding the mathematics, you can ask for help from the lab-assistants. Put effort on understanding the mathematics as much as you can, but do not spend too much time (i.e. it is not the focus of this lab).

6 Reporting

You need to do demonstration of your program to the la-assistants during the lab session. You need to present your design before starting the implementation. The design should clearly show the threads / processes in the system, what data structures will be used, how the operations are synchronized, and how communication goes between threads / processes.

After successful demonstration, the code and design should be emailed to the lab-assistants for review. Of course the code should be clear, well-structured with proper comments.

In addition, you should submit a report to answer the following questions:

1. How is it possible to run multiple threads / processes on a processor? Explain.
2. What happens if we add one more thread in the server, with a higher priority than other threads, and this new thread run an infinite loop without sleep? How are other processes in the system affected? Why does it behave like this?
3. In your solution, you used *Critical Section* functions to protect some critical sections. What would happen if they are not used? Give a sequence of events that result in a serious error.
4. In Windows, CPU time is directly distributed on a basis of thread. However, in UNIX CPU time is shared based on process, and the time capacity of each process is further distributed on a basis of thread. Assume that you do not believe that the assertion is true for Windows! How could you verify this? Describe how you could do this experiment (in theory)

We also want you to write down the problems you had in the implementation of this lab, and attach this to the above email. (This information will be used to improve the lab by next year.)

7 Tips

- Start by sorting out the concepts and ensure that you really understand what to do. The best way to verify this is by making a design and ask for feedback on this from the lab-assistant.
- Do not do everything at once. One tip is to write a standalone application that simulates planetary motion and print them. Then copy the code into the "real" simulator, add graphics support and Try Again. Then it might be helpful to target the communication.

A System Calls in Windows

This appendix provides brief information on some of the system calls that Windows provides, through the so-called Win32 API. More detailed information about these calls are available on MSDN. Here we only briefly mention some calls, and then give an example of how these can be used.

In Win32, the following functions used for managing mutual exclusion and semaphores:

1. Mutual exclusion in a single process: InitializeCriticalSection, EnterCriticalSection, Leave Critical Section, TryEnterCriticalSection.
   Example:

   Take a piece of a global variable, e.g. 'CRITICAL_SECTION foo;'
   Initiate Section: 'InitializeCriticalSection (& foo); "(done once)
   Go into the critical section:' Enter Critical Section (& foo);"
   Exit critical section: 'Leave Critical Section (& foo); "

2. Mutual exclusion between processes: OpenMutex, CreateMutex, ReleaseMutex, Wait - ForSingleObject, Close Handle.

3. Semaphores that can be shared between processes: OpenSemaphore, CreateSemafore, ReleaseSemaphore, WaitForSingleObject, Close Handle.

To change the scheduling priority on processes and threads, you can use: SetThreadPriority, GetThreadPriority, SetThreadPriorityBoost, GetThreadPriorityBoost, GetProcessPriority-Boost, SetProcessPriorityBoost, SetPriorityClass, and GetPriorityClass. (The priorities are initialized during the creation of processes, but can be changed with these functions.)

See MSDN for more information on system calls in Windows!

A Linux threading functionality

From lab 1, you should have gotten familiar with pthreads and sockets which provide threading and inter-process communication. However, since the inter-process communication for Linux in this lab works a little differently, we will create mutexes instead of critical sections.

```
pthread_mutex_t  //Basic declaration of mutex
pthread_mutex_lock //Locking functionality of mutex
pthread_mutex_unlock //Unlocking functionality of mutex
pthread_mutex_destroy//Delete mutex
```

You can find the full thread documentation at https://computing.llnl.gov/tutorials/pthreads/