Step 14: Dynamic Memory Allocation and Data Structures

This assigment is done in class on (monday)04-21-14 AND (friday)04-18-14.

This assignment is due for monday class 04-21-14 AND for the firday class 04-18-14.

Resources

Please refer to this section for resources to help complete the step.

- The RobotLib Library Documentation

Getting Started

Today we will be working with a *robot factory floor simulator*. We have six machines on the factory floor. Each machine can request the robot and the robot will go to the machine. In the program you click on a machine to simulate a request for the robot. Right now the robot is not very fair, since it will always move to the latest request, overriding any previous requests. We are going to fix that.

This project must be compiled and executed on a machine with an X-Server and wxWidgets. This is true for the Linux lab machines. I make no guarantee about personal machines and it is unlikely that you can work on this project remotely given the animation it uses. It is best to work on this project in the lab.

First, create a directory named step14 in your Linux account. You can put this under cse251 or under the root directory, wherever you want to put it. Remember, you use the mkdir command to create a directory:

```
mkdir step14
cd step14
```

Then download the file Robot.tar.gz into that directory. You can right click on the link in the browser and choose Save Link As... to save the file.

Then, using a terminal window in the directory where you put Robot.tar.gz, do this command:

```
tar xvzf Robot.tar.gz
```

This will create a directory called **Robot**. Change into that directory using cd:

```
cd Robot
```

Please edit the file robot.c and add your name to it in a comment.

Compiling and Running the Project

To compile the project, use the make command:

```
make robot32
```

for 32 bit machines or

```
make robot64
```

for 64 bit machines (See Project 2 for details)

You should see output something like this:

```
cbowen@ubuntu:~/cse251/RobotAssignment/Robot$ make
gcc -c -o robot.o robot.c
g++ -o robot robot.o -lm libRobotLib.a `wx-config --libs`
```

This is building your application for you. Whenever you need to recompile, just type make.

To run the program for the first time, type:

```
./robot
```

When you run this program you will see a robot in the middle of the window. Clicking on one of the six machines will cause the robot to move to the machine. Machines in the program are numbered 1 to 6. You will see a message like this when you click on a machine:

```
cbowen@ubuntu:~/cse251/RobotAssignment/Robot$ ./robot
Robot System Startup
Robot request from machine 2
```

Play with the program a bit so you understand how it works and what it does.

There is online documentation for the RobotLib simulator available.

1. Understanding the Program

Edit the program file robot.c. This file looks like this:

```
#include <stdio.h>
#include <stdlib.h>

#include "RobotLib.h"

void OnMachineRequest(int machine);

/*
 * Name :        <insert name here>
 * Description : Robot scheduler program
 */

/*
 * Name :        main()
 * Description : Program entry point.
 */
int main()
{
    /*
```

```
     * This call starts the system running
     */
    printf("Robot System Startup\n");
    SimulatorStartup();

    SetMachineRequestHandler(OnMachineRequest);

    /*
     * This loop runs until we shut the simulator down
     * by closing the window it runs in.
     */
    while(IsSimulatorRunning())
    {



    }

    /*
     * This call shuts down the elevator system
     */
    printf("Robot System Shutdown\n");
    SimulatorShutdown();
    return 0;
}

void OnMachineRequest(int machine)
{
    printf("Robot request from machine %d\n", machine);
    RobotGoTo(machine);
}
```

The program works much like the previous Elevator and Garage projects. There is a control loop that runs until you shut the program down. New in this program is a function OnMachineRequest that is called when a machine requests the robot. You send the robot to a machine using the RobotGoTo function. As you can see, every request immediately sends the robot to the requesting machine. You click on machines in the simulator to issue a request.

A Pending Request

A problem with this program as it stands is that requests get ignored if a new request comes along. This is easy to see if you click on a different machine when the robot is going to a machine. The robot immediately turns to the new requesting machine. This is not very fair.

Suppose we maintain a pending request in some way. Add this global variable to the program before the **main** function:

```
int request = 0;
```

We will use this variable to keep track of any robot requests. A value of zero will mean no requests.

Now change the function OnMachineRequest to utilize this varaible, removing the existing RobotGoTo call:

```
void OnMachineRequest(int machine)
{
    printf("Robot request from machine %d\n", machine);
    request = machine;
}
```

Now, in the control loop we need to determine if the robot is available (not busy) AND we have a request. If so, tell the robot to service that request. Add this code to the control loop:

```
    /*
     * This loop runs until we shut the simulator down
     * by closing the window it runs in.
     */
    while(IsSimulatorRunning())
    {
        if(!IsRobotBusy() && request != 0)
        {
            RobotGoTo(request);
            request = 0;
        }
```

```
    }
```

The function IsRobotBusy returns true if the robot is moving. So, !IsRobotBusy will be true if the robot is NOT busy. If the robot is not busy and the request is not equal to zero, the robot can service the request. Note that we clear the request after we tell the robot to service it by setting it to zero, our indication of no current request.

A Better Solution: Creating a Queue

We want to maintain a *queue* of pending requests. A queue is a line. If the robot is heading for machine 4 and machines 3, 5, and 2 request service in that order, it should move to machine 3, then to 5, and finally to 2 after it is done with 4. A queue keeps track of the fact that we need to service machines 3, 5, and 2 after we are done servicing the current machine.

We have no idea how big the queue will get, so we will have to dynamically allocate it. We are going to use a data structure called a *linked list* to create our queue. It is an efficient way to keep track of information like this.

Add this struct to your program before the main function:

```
struct Request
{
    int machine;
};
```

This structure holds one machine request. Now, add these two variables, also global, before the main function, but after your definition of the Request struct:

```
struct Request *firstInLine = NULL;
struct Request *lastInLine = NULL;
```

We will keep track of a request that is first in line and a request that is last in line. We have pointers to both ends of the line. Right now the line is empty, so these pointers are NULL.

Now change the function OnMachineRequest to allocate a struct of type Request and make it the first and last in the line:

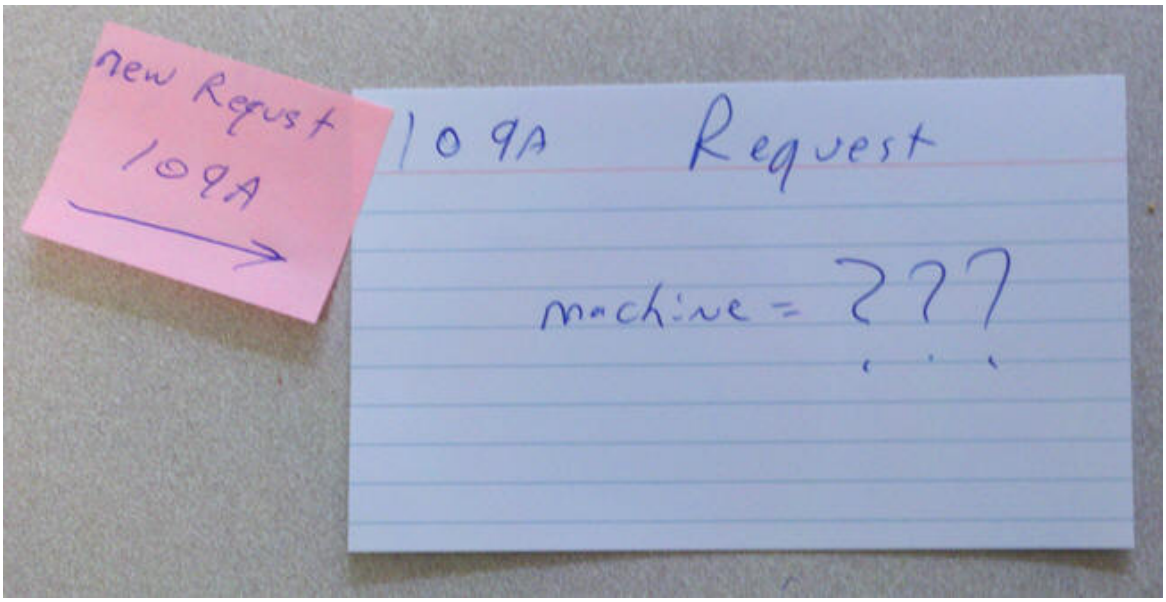
```
void OnMachineRequest(int machine)
{
    printf("Robot request from machine %d\n", machine);
    struct Request *newRequest = malloc(sizeof(struct Request));
    newRequest->machine = machine;

    firstInLine = newRequest;
    lastInLine = newRequest;
}
```

Here is one way to look at what this is doing: This line of code creates a place in memory large enough to hold our Request struct and makes the variable newRequest point to it:

```
    struct Request *newRequest = malloc(sizeof(struct Request));
```

Imagine that spot in memory as a notecard. It has an address. I don't know what it will be, but let's assume it is address 109a (Hex). This is what the situation looks like after that line of code:

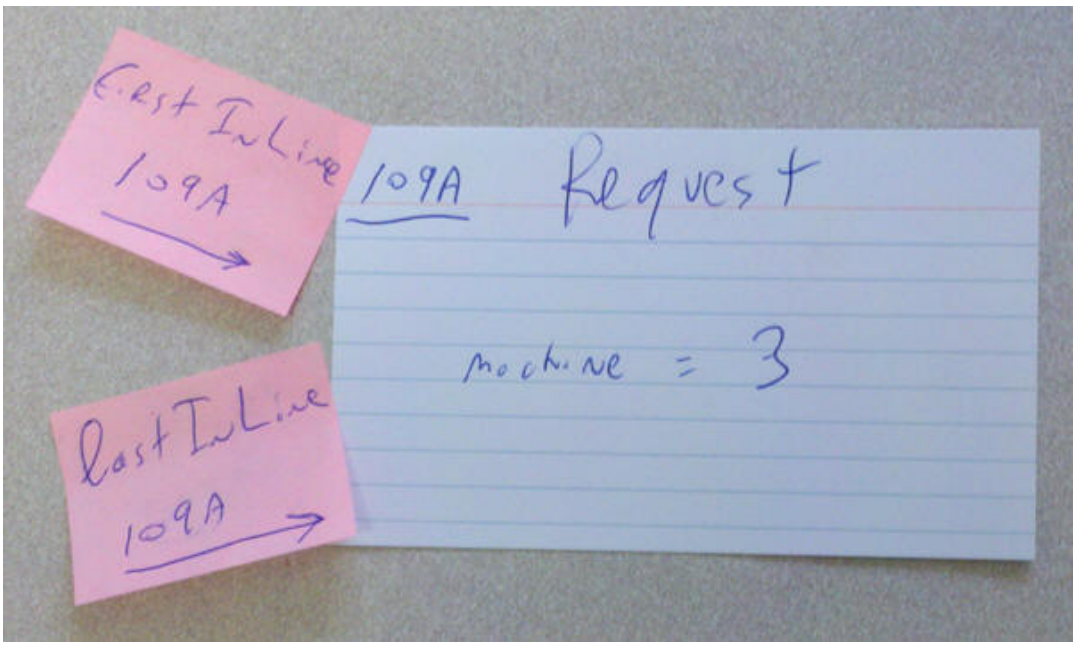


The next three lines of code set the machine number and the other pointers firstInLine and lastInLine to point to this request:

```
newRequest->machine = machine;

firstInLine = newRequest;
lastInLine = newRequest;
```

Let's assume for a moment that the request is from machine 3. Then we have the following situation when the function ends. We have a Request structure in memory at address 109a. We have two pointer variables, both with the value 109a, so they both point to this request.



Now we can change the control loop to use this value:

```
/*
 * This loop runs until we shut the simulator down
 * by closing the window it runs in.
 */
while(IsSimulatorRunning())
{
    if(!IsRobotBusy() && firstInLine != NULL)
    {
        RobotGoTo(firstInLine->machine);

    }

}
```

This will compile and run just like before. We've not yet made it really a queue, only an allocated way to keep track of one request.

To make this a linked list, we need to add a pointer to struct Request that points to the next request in line. Add this line to struct Request:

```
struct Request
{
    int machine;
    struct Request *next;
};
```

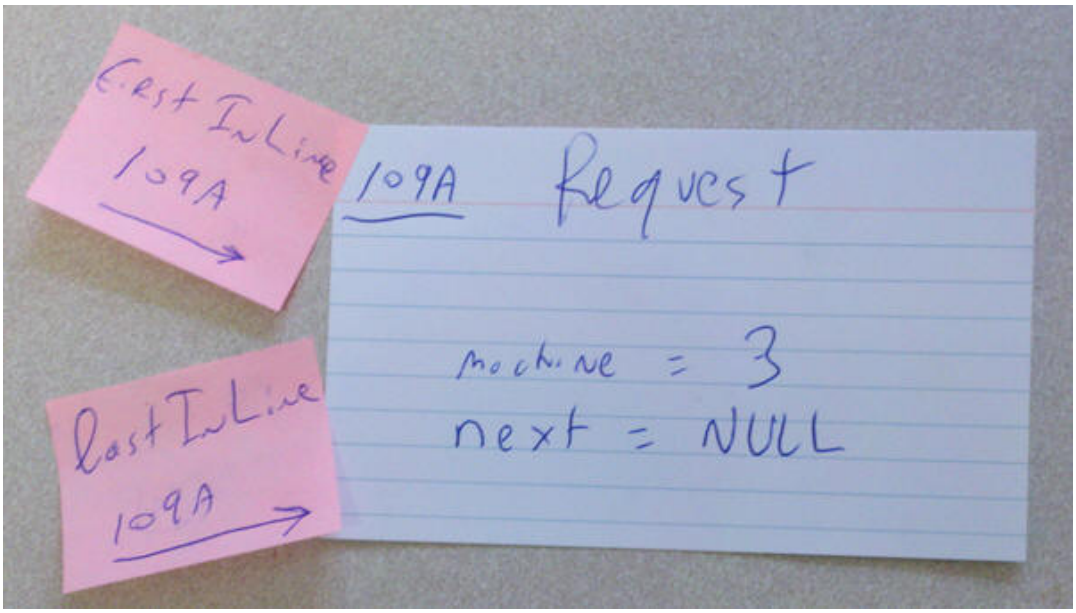
The idea is that every request knows the next request in line. The last request in line will have a next value of NULL.

We need to ensure we set next to NULL for now. Add this line to OnMachineRequest:

```
void OnMachineRequest(int machine)
{
    printf("Robot request from machine %d\n", machine);
    struct Request *newRequest = malloc(sizeof(struct Request));
    newRequest->machine = machine;
    newRequest->next = NULL;

    firstInLine = newRequest;
    lastInLine = newRequest;
}
```

When the first request comes along, it allocates a struct Request in memory. It sets the machine member to the requesting machine and it sets the next pointer to NULL. Because it is both the first in line and the last in line, both firstInLine and lastInLine point to this one request. The value of next is NULL, indicating there is no next request. In our notecard analogy, this is what it looks like:



But, what if a second request comes along? It should become the new last in line and the what was previously the last in line should point to it. If there is no lastInLine, we do what we have been doing: create the one request and make the the first and last in line. But, if there is a lastInLine, we want to make it point to this new request. Replace this line in OnMachineRequest:

```
    firstInLine = newRequest;
```

With this code:

```
    if(lastInLine != NULL)
    {
        /* This is the new last in the line */
        lastInLine->next = newRequest;
        lastInLine = newRequest;
    }
    else
    {
        /* The queue is empty, make this the first request */
        firstInLine = newRequest;
    }
```

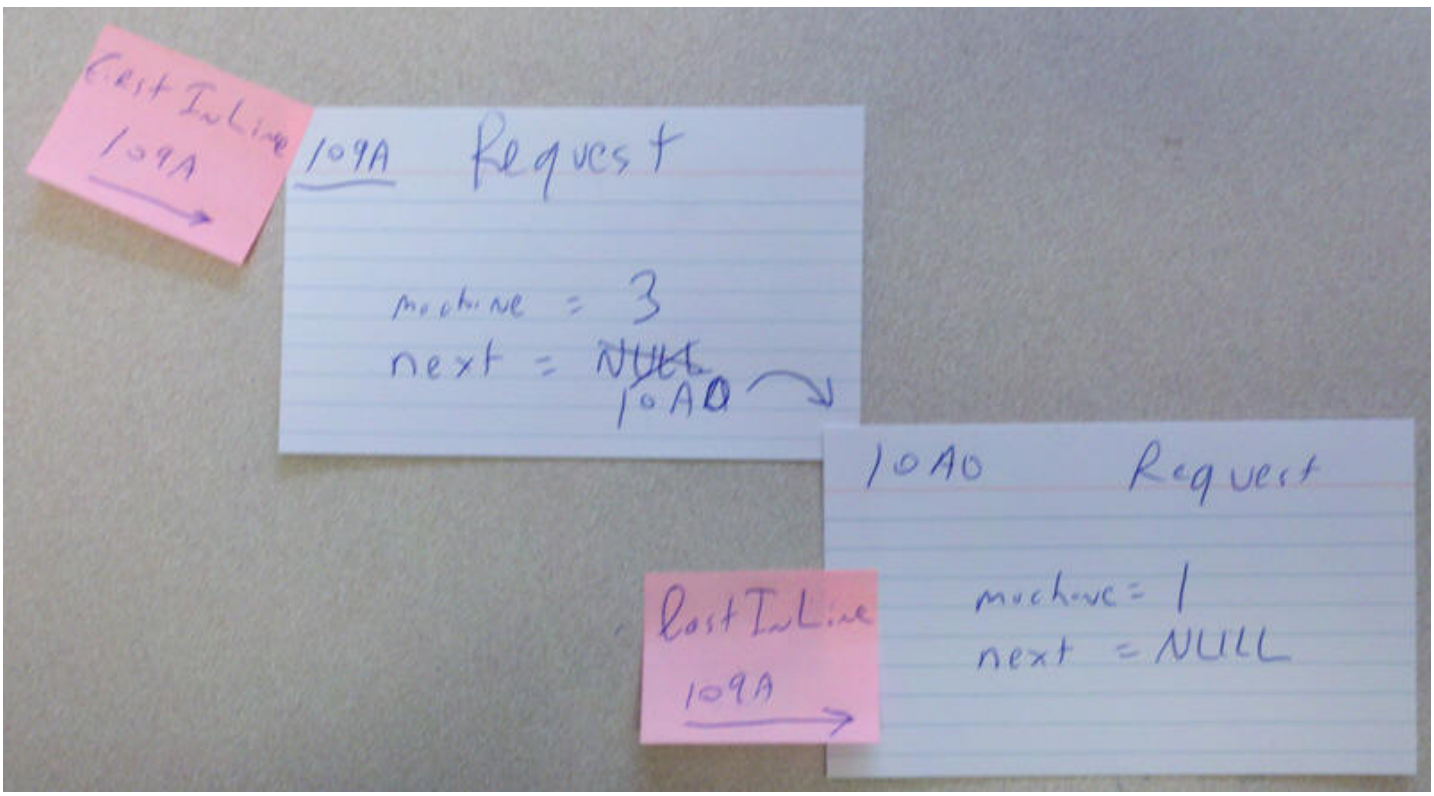
The entire OnMachineRequest function will look like this:

```
void OnMachineRequest(int machine)
{
    printf("Robot request from machine %d\n", machine);
    struct Request *newRequest = malloc(sizeof(struct Request));
    newRequest->machine = machine;
    newRequest->next = NULL;

    if(lastInLine != NULL)
    {
        /* This is the new last in the line */
        lastInLine->next = newRequest;
        lastInLine = newRequest;
    }
    else
    {
        /* The queue is empty, make this the first request */
        firstInLine = newRequest;
    }

    /* The new request is already the last in line */
    lastInLine = newRequest;
}
```

Suppose a second request comes along from machine 1. It will create a new Request object. In our notecards analogy, let that card have address 10a0. If there is a last in line, meaning someone is in line at all, we change the next value in the last in line to point to our new card. Then we change the lastInLine to also point to the new card, which is the new last in line since it is the last request to arrive. Here is what this looks like:



Removing Things from the Queue

We are still not removing things from the queue when we service them. Add this code to the control loop in the main function:

```
    while(IsSimulatorRunning())
    {
        if(!IsRobotBusy() && firstInLine != NULL)
        {
            RobotGoTo(firstInLine->machine);

            /* Remove the first item from the queue */
            struct Request *wasFirst = firstInLine;
            firstInLine = firstInLine->next;
```

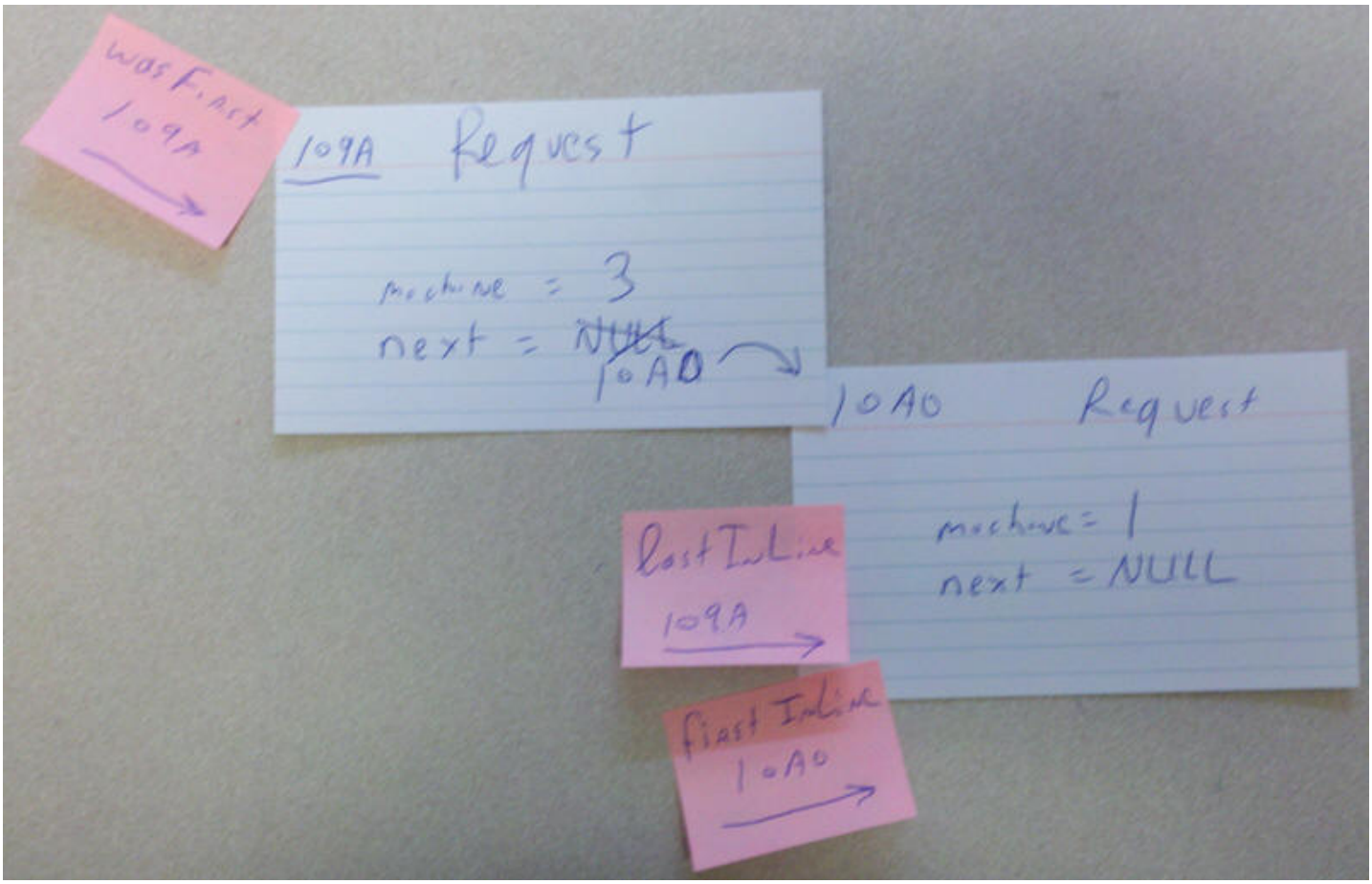
```
            /* Handle an empty queue */
            if(firstInLine == NULL)
                lastInLine = NULL;

        }

    }
```

What this does: If we have a current firstInLine, we tell the Robot to go to it. We keep a pointer to the firstInline in the variable wasFirst. The value of firstInLine->next is the next request in line. That becomes the new firstInLine. Of course, there may not be another request if we get to the end of the list. If there is no next in line, firstInLine->next will be equal to NULL, so firstInLine will be set to NULL. If that is the case, not only do we not have a first in line, we also don't have a last in line because the line is empty.

This is an illustration of what this is doing:



One last thing we need to do here: When we allocate memory with malloc, we need to also free it. Add this line of code to free the memory after we have serviced the request:

```
    while(IsSimulatorRunning())
    {
        if(!IsRobotBusy() && firstInLine != NULL)
        {
            RobotGoTo(firstInLine->machine);

            /* Remove the first item from the queue */
            struct Request *wasFirst = firstInLine;
            firstInLine = firstInLine->next;

            free(wasFirst);

            /* Handle an empty queue */
            if(firstInLine == NULL)
                lastInLine = NULL;

        }

    }
```

Traversing a Linked List

Every time you add something to the queue, I would like to know how many pending requests there are, something like this:

```
cbowen@ubuntu:~/cse251/RobotAssignment/Robot$ ./robot
Robot System Startup
Robot request from machine 2
There are 1 pending requests
Robot request from machine 1
There are 1 pending requests
Robot request from machine 6
There are 2 pending requests
Robot request from machine 5
There are 2 pending requests
Robot request from machine 4
There are 2 pending requests
Robot request from machine 2
There are 3 pending requests
```

First, create an empty function called CountRequests:

```
int CountRequests()
{


    return 0;
}
```

Then add this line at the end of the function OnMachineRequest:

```
    printf("There are %d pending requests\n", CountRequests());
```

Right now if you run this it will always say there are zero pending requests. We need to implement the function CountRequests.

To count the requests we will need to walk along the list. Look at the first request in line and count it. Then look at the next request in line. Repeat that until we are done. Add these two variables to CountRequests:

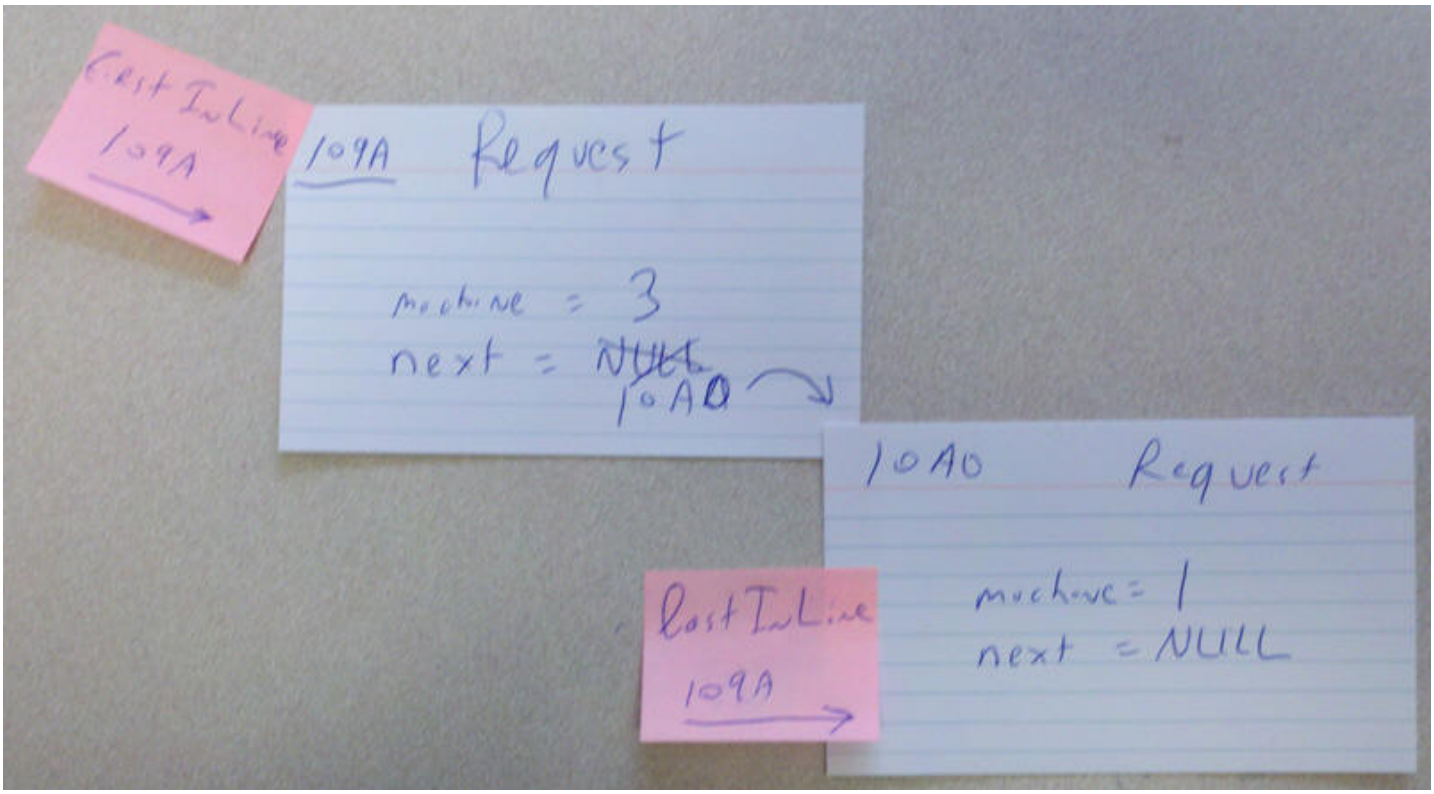
```
    int cnt = 0;
    struct Request *request = firstInLine;
```

Change CountRequests to return the value of cnt:

```
int CountRequests()
{
    int cnt = 0;
    struct Request *request = firstInLine;


    return cnt;
}
```

You have a variable named request. It is pointing to the first request in the list. As long as it is not NULL, it is pointing to a valid request. So, while this variable is not equal to NULL, we want to increment cnt by one and set request to the next request in the list. Notice: we are reading the list, not destroying it, so we do not call free or malloc here. Keep in mind what the value of next in the structure means and this should just require a small loop to complete. Here is the situation with exactly two requests:



This example should output:

```
Robot request from machine 1
There are 2 pending requests
```

Keep in mind that you are also handling requests, so a request for machine 3 may be immediately handled, so when the request for machine 1 comes along it will be the only request. To duplicate the above example, click on machines 4, 3, and 1 in order. The output should be something like this:

```
cbowen@ubuntu:~/cse251/RobotAssignment/Robot$ ./robot
Robot System Startup
Robot request from machine 2
There are 1 pending requests
Robot request from machine 3
There are 1 pending requests
Robot request from machine 1
There are 2 pending requests
Robot System Shutdown
```

This should give you enough information to implement CountRequests.

Final Tasks

1. Be sure you completed the CountRequests task.

2. Add code to your program to support the following behavior:

If the robot sits idle at a machine for more than three seconds, it should return to its home. You make it return to its home with this call:

```
RobotGoTo(0);
```

This task will require you to rearrange the code in the control loop a bit. The functions ResetTimer and GetTimer are available in this program and work just like the versions in the Elevator and Garage simulators.

Turn in robot.c via http://secure.cse.msu.edu/handin/

CSE 251