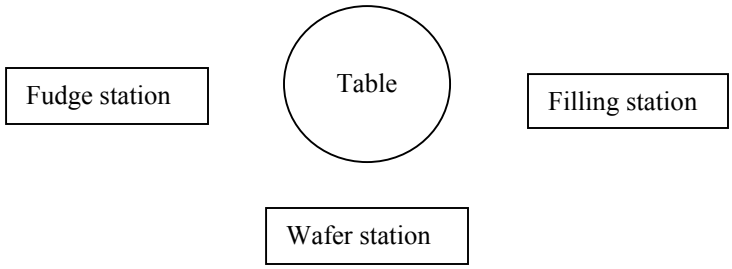
Inside the Cookie Factory – Deadlock Prevention

In this project, you will develop another simulation that uses pthreads. The main focus of this project is to design a solution that will avoid deadlock.

Cookies to Code Simulation

This simulation returns to the cookie-making elves, but this time simulates the assembly of a particular type of cookie that has as ingredients a fudge coating, a wafer, and filling. Three **cookie-assembling elves** stand at three stations (one elf per station) that have infinite amounts of the three ingredients: fudge, wafers, and filling. The stations are arranged in a circle, with a table in the middle as shown below.



In addition, there is one **helper elf** that repeatedly gets two random but different ingredients from the stations and puts them on the table. The helper elf then waits until a cookie has been successfully assembled by one of the cookie-assembling elves. When a cookie-assembling elf finishes a cookie, it rings a bell to notify the helper elf. Upon hearing the signal bell, the helper elf stops waiting, increments a counter of the number of cookies made, and then starts over, getting two new random ingredients and putting them on the table.

The cookie-assembling elves wait at their stations until the two other ingredients they do not have a supply of are on the table. If a cookie-assembling elf obtains the two other ingredients needed, then the elf will assemble a cookie. For example, if the helper elf places fudge and filling on the table, then the elf at the wafer station can assemble a cookie. Similarly, if the helper elf places a wafer and fudge on the table, then the elf at the filling station can assemble a cookie. After assembling a cookie, the cookie-assembling elf will ring the bell to signal the helper elf.

Your implementation must meet all the specifications outlined in this document, including:

- The helper elf must be implemented as shown on the next page. You may need to add additional threads, mutexes, and shared variables to implement this project, but the helper elf CANNOT make use of them.
- Each elf must be implemented as a separate thread using the pthread library as installed on the CSLab Linux machines running Fedora Core 4 in McBryde 124. This means you will have at least four threads: one for each of the cookie-assembling elves and one for the helper elf.
- Deadlock must be avoided.
- Once two ingredients are placed on the table, one of the cookie-assembling elves should (eventually) make a cookie. In other words, there should not be starvation. Because the helper elf always randomly chooses two different ingredients, each time the helper places two ingredients on the table, there will be a combination of ingredients on the table that are needed by one of the cookie-assembling elves to make a cookie.
- The simulation should continue until a pre-specified number of cookies have been made.
- Your solution can only use mutexes and shared variables for synchronization. Specifically, you may not use counting semaphores or condition variables. The only pthread functions you can use in this project are:

```
pthread_create()                pthread_exit()
pthread_mutex_init()            pthread_mutex_lock()
pthread_mutex_unlock()
```

The main challenge in this project is to create a solution that avoids deadlock.

The helper elf must be implemented as indicated by the pseudocode `elf_helper` shown below. Note that there are 6 mutexes used: `m_bell` to represent the signal bell, `m_fudge`, `m_wafer`, and `m_filling` to signal when the helper elf places the corresponding ingredient on the table, `m_sim_done` to signal to the parent thread when the simulation is done (i.e. `max_cookies` have been assembled), and `m_helper_stop` is used so that the helper will block (i.e. stop putting ingredients on the table) while the parent thread prints the summary and exits. You may need to add additional threads, mutexes, and/or shared variables to implement this project, but the helper elf cannot make use of them.

elf_helper

```
srand(seed);
while (1)
{
       m_bell.lock()
       increment num_cookies
       print num_cookies message
       if (num_cookies == max_cookies) {
              m_sim_done.unlock()       /* signal that the sim is done */
              m_helper_stop.lock()      /* stop helper processing */
       }

       /* pick first random ingredient */
       random_ingredient1 = select_random_ingredient(NULL);

       /* pick second random ingredient different from the first */
       random_ingredient2 = select_random_ingredient(random_ingredient1);

       /* signal that the ingredients are available */
       switch (random_ingredient1)
              case fudge:
                     m_fudge.unlock(); print fudge; break;
              case wafer:
                     m_wafer.unlock(); print wafer; break;
              case filling:
                     m_filling.unlock(); print filling; break;
       switch (random_ingredient2)
              case fudge:
                     m_fudge.unlock(); print fudge; break;
              case wafer:
                     m_wafer.unlock(); print wafer; break;
              case filling:
                     m_filling.unlock(); print filling; break;
}
```

Suppose the cookie-assembling elves were implemented to run the code shown below in infinite loops:

elf_fudge	**elf_wafer**	**elf_filling**
m_wafer.lock()	m_filling.lock()	m_fudge.lock()
m_filling.lock()	m_fudge.lock()	m_wafer.lock()
make_cookie()	make_cookie()	make_cookie()
m_bell.unlock()	m_bell.unlock()	m_bell.unlock()

The implementation of the cookie-assembling elves shown above has a number of potential paths that will result in deadlock. For example, suppose that the helper elf places fudge and a wafer on the table. If the elf at the filling station takes the fudge and the elf at the fudge station takes the wafer, then deadlock will occur. The problem is that each cookie-assembling elf is trying to obtain two resources in order to assemble a cookie. Your implementation must avoid deadlock.

Output and Example

Two command-line arguments will be passed to the simulation: the number of cookies to make in this simulation run, and a seed value for the random number generator.

The first command-line argument should be the number of cookies to make in this simulation run. When that many cookies have been made, then the simulation should stop, print a summary indicating the total number of cookies made and how many cookies were assembled by each elf, and then exit.

The second command-line argument should be a seed value for the random number generator. You should use the `srand()` and `rand()` functions in the C standard library `<stdlib.h>`. The second command-line argument should be passed into `srand()` to seed the random number generator.

Your output should be similar to the output shown below:

```
Starting a simulation to make 5 cookies.

elf_helper:  cookies made so far = 0
   putting fudge
   putting filling
elf_wafer:
   made a cookie

elf_helper:  cookies made so far = 1
   putting fudge
   putting filling
elf_wafer:
   made a cookie

elf_helper:  cookies made so far = 2
   putting wafer
   putting fudge
elf_filling:
   made a cookie

elf_helper:  cookies made so far = 3
   putting wafer
   putting filling
elf_fudge:
   made a cookie

elf_helper:  cookies made so far = 4
   putting filling
   putting wafer
elf_fudge:
   made a cookie

elf_helper:  cookies made so far = 5

Summary
   total cookies made  = 5
   cookies by elf_fudge = 2
   cookies by elf_wafer = 2
   cookies by elf_filling = 1

Exiting simulation.
```

Documentation

In addition to writing good in-line documentation in your code, you must also include a file named "`readme`" that contains a brief description of the way you implemented the simulation, how you protected resources, and how you insured there would not be deadlock or starvation.

Design and implementation requirements

There are some explicit requirements, in addition to those on the Programming Style page of the course website:

- Your simulation must be implemented in ISO-compliant C/C++ code. You may make use of any language-standard types and/or containers you find useful.
- You must decompose your implementation into separate source and header files, in some sensible manner that reflects the logical purpose of the various components of your design.
- You must document your implementation according to the *Programming Standards* page on the course website.
- You must properly allocate and de-allocate memory, as needed.

In general, you are expected to apply the design and implementation guidelines and skills covered in your previous computer science courses. You may implement your solution in pure C, or in C++.

Evaluation

Remember that your implementation will be tested in the McBryde 124 lab environment (GCC/Linux). No help will be given for any other development environments. It is your responsibility to make sure your implementation works correctly in the lab.

Your implementation will be evaluated for documentation and design, as well as for correctness of results. Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

Your simulation should be named `cookieassembly` and should accept two command-line arguments: the number of rounds to execute, and a seed for `srand()`. For example:

```
bash> cookieassembly 25 1234
```

should cause your simulation to run for 25 rounds with a random number seed of 1234.

What to turn in and how

This assignment will be collected on the Curator system. The testing will be done under the McBryde 124 lab environment. You may write your code in either C or C++ for this project.

Submit a single gzip'd tar file containing the C/C++ source and header files, the readme files, and the test case files for your implementation to the Curator system. Submit nothing else. Be sure that your header files only contain `include` directives for Standard C/C++ and UNIX header files; any other `include` directives will probably cause compilation errors. It must be possible to unpack the file you submit using the following command:

```
tar –zxf <name of your file>
```

The unpacked files will then be compiled, typically using the following command syntax:

```
[g++ | gcc] –o <name we give the executable> [*.cpp | *.c] -lpthread
```

The appropriate link for submitting to the Curator system will be posted on the course web site. You will be allowed to submit your project up to three times before the due date in order to fix errors you discover before your project is tested. Your latest submission will be the one that is tested.

Pledge

Your program submission must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the pledge statement provided on the course web site in one of your submitted files.