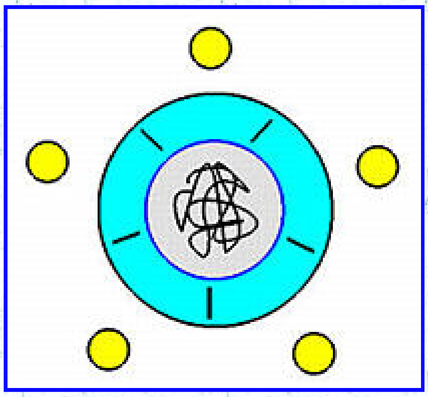
ThreadMentor: The Dining Philosophers Problem

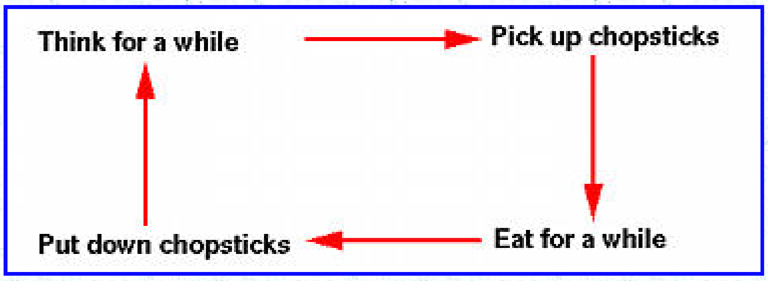
Problem

The ***dining philosophers problem*** is invented by E. W. Dijkstra. Imagine that five philosophers who spend their lives just thinking and easting. In the middle of the dining room is a circular table with five chairs. The table has a big plate of spaghetti. However, there are only five chopsticks available, as shown in the following figure. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up ***both*** chopsticks, he eats for a while. After a philosopher finishes eating, he puts down the chopsticks and starts to think.

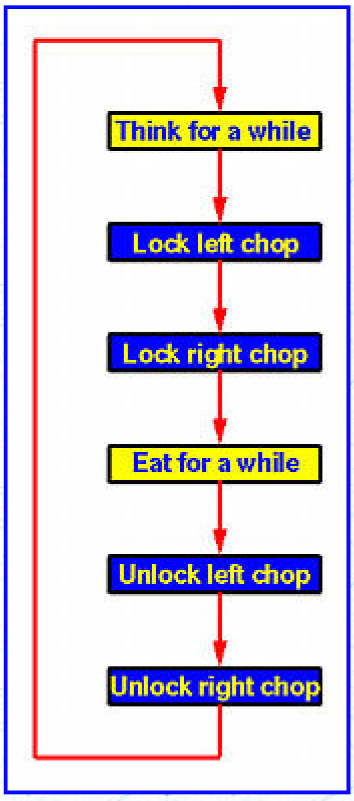


Analysis

How do we write a threaded program to simulate philosophers? First, we notice that these philosophers are in a thinking-picking up chopsticks-eating-putting down chopsticks cycle as shown below.



The "pick up chopsticks" part is the key point. How does a philosopher pick up chopsticks? Well, in a program, we simply print out messages such as ``Have left chopsticks", which is very easy to do. The problem is each chopstick is shared by two philosophers and hence a ***shared*** resource. We certainly do not want a philosopher to pick up a chopstick that has already been picked up by his neighbor. This is a ***race condition***. To address this problem, we may consider each chopstick as a shared item protected by a mutex lock. Each philosopher, before he can eat, locks his left chopstick and locks his right chopstick. If the acquisitions of both locks are successful, this philosopher now owns two locks (hence two chopsticks), and can eat. After finishes easting, this philosopher releases both chopsticks, and thinks! This execution flow is shown below.



Because we need to lock and unlock a chopstick, each chopstick is associated with a mutex lock. Since we have five philosophers who think and eat simultaneously, we need to create five threads, one for each philosopher. Since each philosopher must have access to the two mutex locks that are associated with its left and right chopsticks, these mutex locks are global variables.

Program

Let us see how the above analysis can be convert to a program. Since each philosopher should be run as a thread, we define a `Philosopher` class as a derived class of class `Thread`.

```
#include "ThreadClass.h"

#define PHILOSOPHERS     5

class Philosopher: public Thread
{
     public:
          Philosopher(int Number, int iter);
     private:
          int No;
          int Iterations;
          void ThreadFunc();
};
```

Click here to download this file (`Philosopher.h`)

The constructor takes two arguments, `Number` for the number assigned to this philosopher thread, and `iter` for specifying the number of thinking-eating cycles.

The implementation of this class, as shown below, should have all thinking, eating, locking and unlocking mechanisms implemented. Since each chopstick must be protected by a mutex lock, we declare an array `Chopstick[ ]` of pointers to `Mutex`. Since the main program allocates these locks, they are declared as global variables using `extern`.

Function `Filler()` generates a `char` array that contains some spaces. Note that this function is declared to be `static` so that is can only be used *within* this file (*i.e.*, `Philosopher.cpp`).

Let us look at the constructor. It receives two arguments. The first, `Number`, is assigned by the main program to indicate which philosopher this thread represents. The second, `iter`, gives the number of thinking-eating cycles for each philosopher. The constructor is very simple. It gives this thread a name. Thus, if the value of `Number` is 2 (*i.e.*, philosopher 2), this thread will have the name `Philosopher2`.

```
#include <iostream>
#include "Philosopher.h"

extern Mutex *Chopstick[PHILOSOPHERS];  // locks for chopsticks

static strstream *Filler(int n)
{
     int  i;
     strstream *Space;

     Space = new strstream;
     for (i = 0; i < n; i++)
          (*Space) << ' ';
     (*Space) << '\0';

     return Space;
}

Philosopher::Philosopher(int Number, int iter)
                        : No(Number), Iterations(iter)
{
     ThreadName.seekp(0, ios::beg);
     ThreadName << "Philosopher" << Number << '\0';
}

void Philosopher::ThreadFunc()
{
     Thread::ThreadFunc();
     strstream *Space;
     int i;

     Space = Filler(No*2);

     for (i = 0; i < Iterations; i++) {
          Delay();                                      // think for a while
          Chopstick[No]->Lock();                        // get left chopstick
          Chopstick[(No+1) % PHILOSOPHERS]->Lock();     // gets right chopstick
          cout << Space->str() << ThreadName.str()
               << " begin eating." << endl;
          Delay();                                      // eat for a while
          cout << Space->str() << ThreadName.str()
               << " finish eating." << endl;
          Chopstick[No]->Unlock();                      // release left chopstick
          Chopstick[(No+1) % PHILOSOPHERS]->Unlock();   // release right chopstick
     }
     Exit();
}
```

Click here to download this file (`Philosopher.cpp`)

The function `ThreadFunc()` implements the executable code of a philosopher thread. First of all, it creates a `char` array of `No*2` spaces so that this thread's output would be indented properly. After this, this thread iterates `Iteration` times. In each cycle, this thread simulates thinking and eating. To this end, we use a method of class `Thread`: `Delay()`. The purpose of `Delay()` is simply delaying the execution of the thread for a random number of times. This simulates ``think for a while" and ``eat for a while."

Let us look at how locking and unlocking of chopsticks is carried out. Suppose the chopsticks are numbered counter clockwise. For philosopher *i*, his left chopstick is *i* and his right chopstick is *i*+1. Of course, we cannot use *i*+1 directly, because when *i*=4, the right chopstick of philosopher is 0 rather than 5. This can easily be done with the remainder operator: (*i*+1) % `PHILOSOPHERS`, where `PHILOSOPHERS` is the number of philosophers. In the above code, philosopher `No` thinks for a while, locks his left chopstick by calling the method `Chopstick[No]->Lock()`, locks his right chopstick by calling the method `Chopstick[(No+1) % PHILOSOPHERS]->Lock()`, eats for a while, unlocks his left chopstick by calling the method `Chopstick[No]->Unlock()`, and unlocks his right chopstick by calling the method `Chopstick[(No+1) % PHILOSOPHERS]->Unlock()`. This completes one thinking-eating cycle. This cycle repeats for `Iteration` number of times.

Note that in the above code each philosopher picks up, or locks, his left chopstick first followed by the right one.

The main program, as usual, is easy as shown below. The number of thinking-eating cycles a philosopher must perform is the only command line argument. ***Since mutex locks must be created before their uses, the main program allocates the mutex locks before the creation of threads.*** In the following, each mutex lock is created with a name like ChopStick0, ChopStick1, ..., ChopStick4. After all chopstick locks are created, the main thread continues to create philosopher threads and joins with all of its child threads. When all philosopher threads terminate, the main thread returns (*i.e.*, terminates).

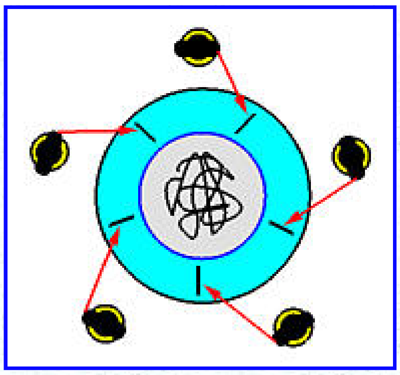
```
#include <iostream>
#include <stdlib.h>

#include "Philosopher.h"

Mutex *Chopstick[PHILOSOPHERS];  // locks for chopsticks

int main(int argc, char *argv[])
{
     Philosopher *Philosophers[PHILOSOPHERS];
     int i, iter;
     strstream name;

     if (argc != 2) {
          cout << "Use " << argv[0] << " #-of-iterations." << endl;
          exit(0);
     }
     else
          iter = abs(atoi(argv[1]));

     for (i=0; i < PHILOSOPHERS; i++) {  // initialize chopstick mutex locks
          name.seekp(0, ios::beg);
          name << "ChopStick" << i << '\0';
          Chopstick[i] = new Mutex(name.str());
     }

     for (i=0; i < PHILOSOPHERS; i++) {  // initialize and run philosopher threads
          Philosophers[i] = new Philosopher(i, iter);
          Philosophers[i]->Begin();
     }

     for (i=0; i < PHILOSOPHERS; i++)
          Philosophers[i]->Join();

     Exit();

     return 0;
}
```

Click here to download this file (Philosopher-main.cpp)

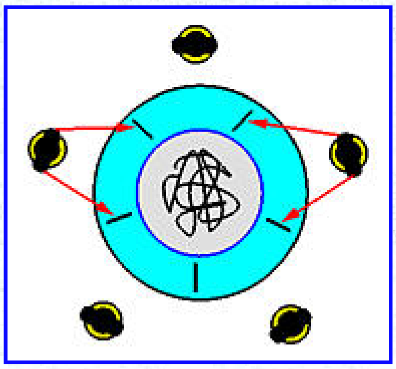
Discussion

Here are some very important facts about this program:

1. If you read the program carefully, we implicitly assign philosopher No to use chopstick ChopStick[No] and chopstick ChopStick[(No+1) % PHILOSOPHERS]. In other word, each philosopher is assigned to a ***fixed*** chair. Is it necessary? It is certainly not. For example, when a philosopher is hungry, we can generate a random integer *i* in the range of 0 and 4. If that chair is occupied, generate another random integer. In this way, we simulate the activity of finding an un-occupied chair. Once an un-occupied chair, say *i*, is found, this philosopher uses chopstick *i* and (*i* + 1) % PHILOSOPHERS. In doing so, our program may be very complex and blur our original focus. After you understand the above program, you can certainly try to make it more realistic.
2. The above program forces each philosopher to pick up and put down his left chopstick, followed by his right one. This is also for the purpose of simplicity. In fact, it is easy to see that the order of putting down the chopsticks is irrelevant. Try to reasoning about this yourself.
3. ***The most serious problem of this program is that deadlock could occur!***
   What if every philosopher sits down about the same time and picks up his left chopstick as shown in the following figure? In this case, all chopsticks are locked and none of the philosophers can successfully lock his right chopstick. As a result, we have a circular waiting (*i.e.*, every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.



4. ***Starvation is also a problem!***
   Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a ***starvation***. Note that it is not a deadlock because there is no circular waiting, and every one has a chance to eat!

   The above shows a simple example of starvation. You can find more complicated thinking-eating sequence that also generate starvation. Please try.