Dining Philosophers Documentation

CSInParallel Project

July 19, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

In this module, you will learn about the Dining Philosophers Problem and some of the solutions that have been developed for it. This is a classic problem in concurrency that describes a situation where multiple processes contend for shared resources. The problem is originally based on an examination question given by Edsger Dijkstra in 1965.

Dining Philosophers source file: `dining_philosophers_code.tar.gz`

1.1 Learning Objectives

- Understand the Dining Philosophers Problem and the situations that it provides a useful model for.
- Understand several solutions for the Dining Philosophers problem and the differences between them.
- Be able to implement a simulation in C or C++ that solves the Dining Philosophers Problem (or at least solves the deadlock portion of the problem).

1.2 Requirements

This module will make use of a number of parallel programming techniques such as threads (through OpenMP) and message passing (through OpenMPI). The code provided with this module has only been tested being compiled with GCC on Linux, although it may work on other Unix systems. GCC has had support for OpenMP since version 4.2, which was released in 2007. Using OpenMPI requires the OpenMPI libraries and headers, which can be found in the software the repositories for many Linux distributions.

In order to run one of the example programs, the OpenMPI installation needs to have support for multiple threads executing in the MPI library concurrently. This requires it to be configured with the `--enable-mpi-thread-multiple` option (or `--enable-mpi-threads` for older versions of OpenMPI). To see if your compilation of OpenMPI supports this feature, the output of `ompi_info | grep Thread` should list **yes** for `MPI_THREAD_MULTIPLE` (or yes for “mpi” threads for older versions of OpenMPI). Note that this only applies to 1 of the 5 example programs.

There are a number of MPI libraries available other than OpenMPI; it should be possible to use a different one.

1.3 Introduction

Suppose that 5 philosophers are sitting around a circular table. A plate of spaghetti is given to each philosopher, and 5 forks are distributed around the table so that each philosopher has a left fork and a right fork that are shared with his neighbors. Each philosopher independently alternates between thinking and eating. In order to eat, a philosopher must acquire both his left and right forks. When done eating, he releases his forks and resumes thinking. Philosophers are allowed to think for an arbitrary amount of time, even an infinite amount of time. Philosophers are allowed to eat for an arbitrary but finite amount of time.

(The problem can also be explained using rice and chopsticks, which perhaps makes more sense, as it is much harder to eat with one chopstick than with one fork! We will stick with forks and spaghetti since it seems to be the more common explanation.)

1.4 Requirements for a Solution

The philosophers, of course, represent processes or threads running in a computer, and the forks represent some kind of shared resource. These shared resources could be a number of things, such as records in a database or files that can be accessed by only one process at a time.

We would like to examine the ways that a solution to the Dining Philosophers problem could be implemented on a real computer or cluster of computers.

A successful solution to the problem must meet the following criteria:

- Each fork is in use by at most one philosopher at any instant in time.
- All philosophers, upon reaching an hungry state, will be able to eat within a finite amount of time.

CHAPTER

TWO

SHARED MEMORY DINING PHILOSOPHERS

In this section, we consider the case where each philosopher is a thread. They all share the same memory space.

2.1 Observing deadlock

Take a look at the file `deadlock.c` in the code distributed with this module. This is a simple shared memory simulation (but not a solution!) of the dining philosophers problem. The process is divided into five threads using an OpenMP parallel section.

```
/* Each philosopher is a thread */
#pragma omp parallel num_threads(NUM_PHILOSOPHERS)
{
        /* Get the number of this thread and figure out which fork is on
         * the right and which is on the left. */
        int thread_num = omp_get_thread_num();
        pthread_mutex_t *left_fork = &forks[thread_num];
        pthread_mutex_t *right_fork = &forks[(thread_num + 1) % NUM_PHILOSOPHERS];

        /* Make this thread have a random seed different from the other
         * threads. */
        unsigned seed = t + thread_num;
        while (1) {
                philosopher_cycle(thread_num, left_fork, right_fork, &seed);
        }
}
```

OpenMP (which stands for Open Multi-Processing) is an API that makes it easier to write multithreaded programs in C, C++, and Fortran. It works mainly by using **pragmas**, directives that give information to the compiler. It also offers a number of functions such as `omp_get_thread_num()`. To use OpenMP in a C or C++ program, you must pass the flag `-fopenmp` to GCC. This flag must be specified both for the compilation phase, so that the OpenMP pragmas are interpreted, and for the linking phase, so that the programs are linked with `libgomp`, the GNU implementation of the OpenMP API (which is part of gcc). Also, if any OpenMP functions are called explicitly, `omp.h` must be included.

The forks are represented by **mutexes**. These are program objects that can only be modified by one thread at a time – that is, modifying them is an atomic operation.

```
/* The forks are represented by an array of mutexes. */
pthread_mutex_t forks[NUM_PHILOSOPHERS];
```

More specifically, a mutex is a special case of a **semaphore**, an object that allows atomic increment and decrement operations. A mutex is a **binary semaphore** that only allows two values. These two states can be referred to as "locked" and "unlocked" A mutex can be used to ensure exclusive access to objects or sections of code because only one thread can have locked the mutex at a time.

Each thread (philosopher) enters a loop in which it thinks and eats.

```
/* Called each time a philosopher wants to eat. */
static void philosopher_cycle(int thread_num, pthread_mutex_t *left_fork,
                pthread_mutex_t *right_fork, unsigned *seed)
{
        printf("Philosopher %d wants to eat!\n", thread_num);

        pthread_mutex_lock(left_fork);
        printf("Philosopher %d picked up his left fork.\n", thread_num);
        pthread_mutex_lock(right_fork);
        printf("Philosopher %d picked up his right fork and "
                "started eating.\n", thread_num);

        sleep_rand_r(min_eat_ms[thread_num], max_eat_ms[thread_num], seed);

        pthread_mutex_unlock(left_fork);
        pthread_mutex_unlock(right_fork);
        printf("Philosopher %d is done eating and has released his "
                "forks.\n", thread_num);

        sleep_rand_r(min_think_ms[thread_num], max_think_ms[thread_num], seed);
}
```

A **reentrant** function– that is, a function that returns consistent results when called by multiple threads concurrently– that sleeps the calling thread for a random number of milliseconds is defined to implement the "thinking" and "eating".

Compile `deadlock.c` using `make deadlock` (or just type `make` to compile all the examples at once), then run it. Observe the simulation for a while before interrupting it.

Did you notice any problems? Most likely not, if you ran the simulation for only a short period of time. There is a major problem with the code though. In the function *philosopher_cycle*, try adding a very short delay between the time the philosophers pick up their left fork and the time the philosophers pick up their right fork, like this:

```
pthread_mutex_lock(left_fork);
printf("Philosopher %d picked up his left fork.\n", thread_num);
millisleep(5);
pthread_mutex_lock(right_fork);
```

Now try running the code again. What happens?

Most likely all the philosophers picked up their left fork, at which point they all became permanently **deadlocked** because each philosopher was waiting on each other in turn. All the threads were waiting for a resource that will never be released.

This problem can only occur if 4 of the 5 threads are preempted after picking up their left fork but before picking up their right fork. This is unlikely to occur without a delay between the locks. But even so, the possibility of deadlock is a major flaw in this code. After all, you wouldn't want to use an operating system or program that could lock up at any time!

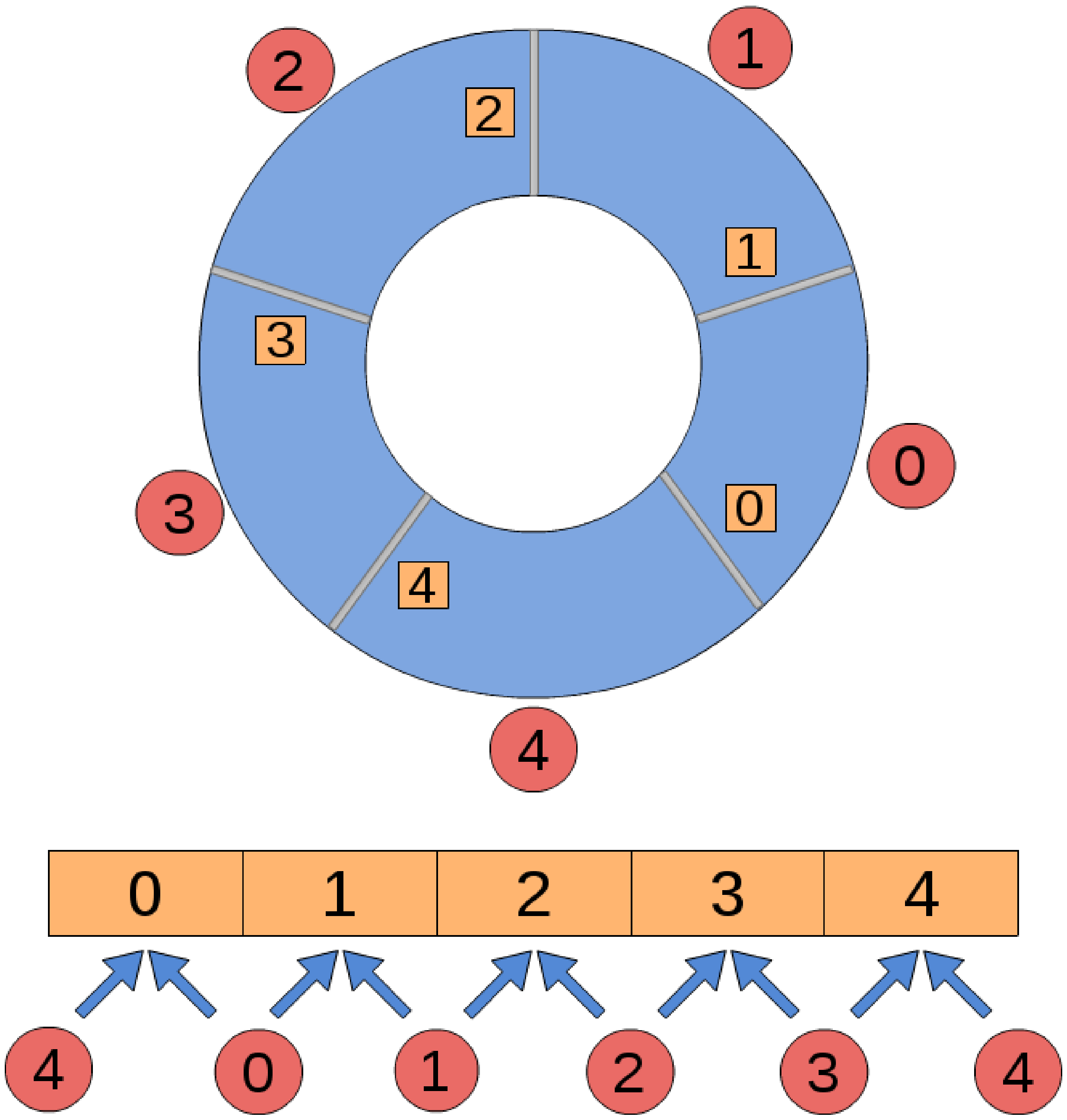

Figure 2.1: Figure 1. In this figure, red circles represent philosophers, or threads. Orange rectangulars represent forks, or common resource that threads are sharing.

2.2 A "Solution" that avoids deadlock

Is there a simple way to prevent deadlock from occurring?

Suppose that not all of the philosophers were to pick up their forks in the same order. That is, some of the philosophers would pick up their forks left-right, and others would pick up their forks right-left. This would therefore be an asymmetrical solution, since not all of the philosophers would act in the same way.

It turns out that deadlock will be avoided simply if the fifth philosopher picks up his forks in the opposite order from everyone else. If the first four philosophers have all picked up their left fork, then the fifth philosopher will be unable to pick up the last fork, since it would be his second fork to pick up. Similarly, if the fifth philosopher is holding one fork, then it is impossible for the first philosopher to pick up any forks. It is therefore guaranteed that deadlock will not occur.

Compile and run `partial_order.c`.

```
if (thread_num == NUM_PHILOSOPHERS - 1) {
        pthread_mutex_lock(right_fork);
        printf("Philosopher %d picked up his right fork.\n", thread_num);
        millisleep(5);
        pthread_mutex_lock(left_fork);
        printf("Philosopher %d picked up his left fork and "
                        "started eating.\n", thread_num);
} else {
        pthread_mutex_lock(left_fork);
        printf("Philosopher %d picked up his left fork.\n", thread_num);
        millisleep(5);
        pthread_mutex_lock(right_fork);
        printf("Philosopher %d picked up his right fork and "
                        "started eating.\n", thread_num);
}
```

It is very similar to `deadlock.c`, but it adds in a simple if-else statement around the code where the philosophers pick up their forks, so that the last philosopher picks up his forks in the opposite order from the others. A delay between picking up the two forks has already been added. You will see that this code will not deadlock.

This solution can be generalized to the idea of assigning a partial order to the resources. In the classic dining philosophers problem, the forks would be numbered from 0 to 4. In this situation, processes must acquire their resources in order from lowest to highest. This will work for any number of processes acquiring any number of resources. However, if the needed resources are not known in advance, this can be an inconvenient solution; this is because if a process has decided it needs a resource after it has already acquired higher numbered resources, it must release the higher numbered resources first, acquire the needed resource, then reacquire the released resources in order.

2.3 Starvation

There is still a problem with the solution in `partial_order.c, however. Take a look at the code in ``partial_order2.c`. This is similar to the original `partial_order.c`, but an interrupt handler has been added. This interrupt handler prints out the number of times each philosopher has eaten when the program is interrupted (when Control-C is pressed). The philosophers also think and eat much faster and have no delay between acquiring the forks.

Compile `partial_order_2.c`, then let it run for a little while before interrupting it. Look at how many times each philosopher has eaten and thought. Are they about the same? Or does it look like some philosophers had an advantage over others?

Your results will vary, but here are our results after running the simulation for a few minutes:

Philosopher	Times Eaten
Philsopher 0:	4341 times eaten
Philsopher 1:	4529 times eaten
Philsopher 2:	4612 times eaten
Philsopher 3:	4673 times eaten
Philsopher 4:	4340 times eaten

Philosopher 4 and 0 appear to eat and think less often than the other philosophers; thus, they spend more time in the "hungry" state, waiting to acquire forks.

This imperfection is a result of the asymmetry of the solution: since not all the philosophers are acting in the same way, it is possible that some philosophers do not have the same chance to eat as others do.

If you think about it, you may realize there is a more fundamental problem as well. Suppose that philosopher 2 wants to eat, but philosophers 1 and 3 are currently eating. Philosopher 2's thread is put to sleep as it waits for the forks. Meanwhile, philosophers 1 and 3 finish eating, but philosopher 2's thread is not scheduled to run right away. Rather, before philosopher 2's thread is run again, philosophers 1 and 3 start to eat again. Philosopher 2 never had a chance to eat! What if this keeps happening over and over?

The problem here is that one of the philosophers could potentially "starve" because of a timing problem. This is independent from the possibility of deadlock, which has already been eliminated in the partial order solution.

We will return to the starvation problem later when we discuss the Chandry-Misra solution.

CHAPTER

THREE

DISTRIBUTED DINING PHILOSOPHERS

3.1 OpenMPI

In this section we discuss the Dining Philosophers problem where each philosopher is a separate process. Memory is not shared. We make use of **OpenMPI**, an implementation of **MPI**, which stands for **Message Passing Interface**. OpenMPI provides an API for sending data from one process to another. Programs using OpenMPI must include `mpi.h` and link with a number of libraries. You can use the wrapper compiler `mpicc` to avoid having to specify all the MPI libraries when linking your program.

MPI programs are run by using the `mpirun` command. This command allows you to specify the number of processes to start and which hosts to start them on. Although the processes may or may not run on different hosts, in the program code itself there is no difference between sending data to a process running on the same computer and sending data to a process running on a different computer. This is because OpenMPI provides the concept of a "communicator" containing a number of processes, where each process has a number. To send a message to a process, you simply need to specify its number and communicator.

OpenMPI supports both blocking and unblocking sends and receives. Blocking sends and receives, such as `MPI_Send()` and `MPI_Recv()`, block until the message is actually send or received, while nonblocking sends and receives, such as `MPI_Isend()` and `MPI_Irecv()`, return immediately. If using nonblocking message passing, you can later test to see if a message has completed using the `MPI_Test()` class of functions or wait for a message to complete using the `MPI_Wait()` class of functions. If you have installed the OpenMPI documentation, there should be a man page on each MPI function.

3.2 A "Waiter" solution to the distributed Dining Philosophers problem

One of the simplest solutions to resource management problems like the Dining Philosophers problem is to centralize the problem by having a master thread or master process that determines which threads or processes are able to access resources. For the Dining Philosophers problem, this can be called the **waiter** solution.

In order to solve the dining philosophers problem in a distributed manner using the waiter solution, the philosopher processes must communicate with the waiter using message passing. That is, they must ask for permission to eat, and the waiter, who has control of all the forks, decides who gets to eat and sends them a message telling them to go ahead.

The waiter may only be concerned with preventing deadlock, or he may also keep track of which philosophers have eaten and give an advantage to philosophers who are starving.

In the file `distributed_waiter.c`, there is an implementation of the waiter solution to the distributed dining philosophers problem. It is not a full solution, since it only solves the deadlock problem and not the starvation problem. Take a look at the code and read the comments to try to understand it. Support for MPI threads is not needed to run this program as it only uses processes, not threads.

Since the waiter is its own process, `distributed_waiter` must be run using 6 processes in order to simulate 5 philosophers.

3.3 The Chandry-Misra solution to the distributed Dining Philosophers problem

Although having a waiter process can fully solve the dining philosophers problem, the disadvantage is that all philosophers have to wait for the one waiter, who could be overloaded with work if there are too many philosophers. Is it possible to solve the dining philosophers problem in a distributed, decentralized manner?

In 1984, K. M. Chandry and J. Misra published a paper titled *The drinking philosophers problem* [1]. In it, they provide a completely distributed solution to the Dining Philosophers problem that avoids both deadlock and starvation. They also generalize their solution to what they call the "Drinking Philosophers problem", where an arbitrary number of agents can share any number of resources ("bottles") with other agents and require any number of these resources for each "drinking" session.

For the full details of their solution, you should see their original paper, which as of this writing is freely available online. We will give you a summary of their solution and show you some code that implements it.

In Chandry and Misra's solution to the Dining Philosophers problem, idea of the forks being *clean* and *dirty* is introduced. The solution is completely distributed, and philosophers must send "request tokens" to other philosophers to request their forks. "Dirty" forks must be given up if they are requested, while "clean" forks may be kept, unless they are not needed. Whenever a fork is used to eat, it becomes dirty, and whenever a fork is sent to another philosopher, it is cleaned (if it is not already clean).

The deadlock problem is solved by ensuring that no cycles can develop in the precedence graph that represents which philosophers have priorities over others. Additionally, the starvation problem is solved because the cleanliness of the forks will ensure that any one philosopher will not wait forever to eat.

In the file `chandry_misra.c`, there is an implementation of this solution. Messages are passed between processes using OpenMPI. Read some of the comments in the code to better understand the solution.

In order to run this program, the MPI library must support multiple threads executing in the MPI library concurrently. This is because each philosopher process is divided into 2 threads: a main thread that does the thinking and eating, and a helper thread that listens for requests for forks from the philosophers' neighbors at all times, including when the main thread is thinking. This design is necessary if we do not allow for the possibility that forks can be owned by no philosophers. If at any point in time, the fork must be held by one of the two philosophers or be in transit between the two, it cannot be guaranteed that the other philosopher can get the fork if one philosopher decides to think for an arbitrarily long period of time, during which he cannot release the fork– but the Chandry-Misra solution requires that he does in fact release the fork somehow. The helper thread solves this problem by allowing the philosopher's forks to be given up while he is thinking.

Run the program with `mpirun -n 5 chandry_misra [SECONDS]`. The default number of seconds to run the simulation for is 5. Statistics are shown when the simulation finishes. The default eating and thinking times are set to be very fast, but you can adjust this in the arrays near the top of the code. Try running the code for a hundred seconds or so and look at the results. Does it look like all the philosophers had an equal chance to eat? You will be able to tell from the results, although they won't be able to tell you if starvation is theoretically possible. But the solution does, in fact, guarantee that each any every philosopher will be able to eat within a finite amount of time once he becomes hungry.

[1]

11. (a) Chandry and J. Misra. The drinking philosophers problem. ACM Transactions on Programming Languages and Systems, 6(4):632–646, October 1984.