CSD204, Operating Systems, Spring 2015

Shiv Nadar University
Instructor: Shashi Prabh

Lab 9, Ungraded: The Dining Philosophers
SGG, Chapter 5

Due: Mar. 25, 2015

1 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again. The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure below:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution-that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

We present a solution later to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

This problem will require implementing a solution using Pthreads mutex locks and condition variables.

The Philosophers
Begin by creating five philosophers, each identified by a number 0...4. Each philosopher will run as a separate thread. Thread creation using Pthreads and mutex locks is already covered in previous two lab assignments. Philosophers alternate between thinking and eating. To simulate both activities, have the thread sleep for a random period between one and three seconds. When a philosopher wishes to eat, she invokes the function

```
pickup_forks(int philosopher_number)
```

where philosopher number identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

```
return_forks(int philosopher number)
```

Pthreads Condition Variables
Condition variables in Pthreads behave similarly to those described in Section 5.8 of your text book. However, in that section, condition variables are used within the context of a monitor, which provides a locking mechanism to ensure data integrity. Since Pthreads are typically used in C programsand since C does not have a monitor we accomplish locking by associating a condition variable with a mutex lock. Pthreads mutex locks are covered earlier. We need to apply Pthreads condition variables here. Condition variables in Pthreads use the `pthread_cond_t` data type and are initialized using the `pthread_cond_init()` function. The following code creates and initializes a condition variable as well as its associated mutex lock:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

The `pthread_cond_wait()` function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition `a == b` to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
```

```
while (a != b)
pthread_cond_wait(&mutex, &cond_var);

pthread_mutex_unlock(&mutex);
```

The mutex lock associated with the condition variable must be locked before the `pthread_cond_wait()` function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes `pthread_cond_wait()`, passing the mutex lock and the condition variable as parameters. Calling `pthread_cond_wait()` releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.) A thread that modifies the shared data can invoke the `pthread_cond_signal()` function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to `pthread_cond_signal()` does not release the mutex lock. It is the subsequent call to `pthread_mutex_unlock()` that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to `pthread_cond_wait()`.

Dining-Philosophers Solution Using Monitors

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher `i` can set the variable `state[i] = EATING` only if her two neighbors are not eating: `(state[(i+4) % 5] = EATING)` and `(state[(i+1)% 5]= EATING)`.

Pseudo code from sec 5.8

```
monitor DiningPhilosophers
{
enum {THINKING, HUNGRY, EATING} state[5];
condition self[5];

void pickup(int i) {
state[i] = HUNGRY;
test(i);
if (state[i] != EATING)
```

```
self[i].wait();
}

void putdown(int i) {
state[i] = THINKING;
test((i + 4) % 5);
test((i + 1) % 5);
}

void test(int i) {
if ((state[(i + 4) % 5] != EATING) &&
(state[i] == HUNGRY) &&
(state[(i + 1) % 5] != EATING)) {
state[i] = EATING;
self[i].signal();
}
}

initialization_code() {
for (int i = 0; i < 5; i++)
state[i] = THINKING;
}
}
```

We also need to declare

```
condition_self[5];
```

This allows philosopher `i` to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `DiningPhilosophers`, whose definition is shown in in the Pseudo code below. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher `i` must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

2 Instructions

- Implement the solution to Dining Philosopher Problem using POSIX API in C. Your program should take number of philosophers from user as input and output the state of a philosopher who is eating. Name your solution source code according to the supplied Makefile.
- Learn to use the supplied Makefile. Use makefile macros to name your compiler and path to your source/object files.
- Upload tar ball using same folder structure on Blackboard before submission deadline.