Programming Project 7

OS2024-SWU-计科中外 34 班

Fall 2024

Experimental topic

Programming Projects in Chapter 7

Page P-39 to P-39 of Operating System Concepts (10th) or this pdf

Experimental resources

/resources/code_for_project/code_for_project7

Experimental report naming format:

StudentID-name-Project-X (e.g., 2220183211-张三-Project-1)

Experimental objectives

1. (Course Objective 2) Master the basic concepts, design ideas and operation status of the process. Understand the basic principles and ideas of process synchronization and mutual exclusion, and identify the key links, steps and constraints of engineering problems in this field. Use the basic means of realizing process synchronization and mutual exclusion to analyze and solve the dining problem of philosophers in the field of process synchronization and mutual exclusion, and learn the basic means to solve the deadlock problem.

Experimental report submission

> **Attention**
>
> Please upload your report with naming format to the ftp server within **two weeks** (Deadline: 2024-12-05).

1 The Dining Philosophers Problem

In Section 7.1.3, we provide an outline of a solution to the dining-philosophers problem using monitors. 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 is covered in Section 4.4.1. 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)
```

2 Pthreads Condition Variables

Condition variables in Pthreads behave similarly to those described in Section 7.3.3. 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 is typically used in C programs—and since C does not have a monitor—we accomplish locking by associating a condition variable with a mutex lock. Pthreads mutex locks are covered in Section 7.3.1. We cover 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);
```

(continues on next page)

(continued from previous page)

```
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()`.