Chapter 7

Synchronization Examples

Chapter 7: Synchronization Examples

- Classic Problems of Synchronization
  - The Bounded-Buffer Problem
  - The Dining-Philosophers Problem

- POSIX Synchronization
  - POSIX Mutex Locks
  - POSIX Semaphores

Classic Problems of Synchronization

- In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems.
  - These problems are used for testing nearly every newly proposed synchronization scheme.

- In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions.

- However, implementations of these solutions could use mutex locks in place of binary semaphores.

The Bounded-Buffer Problem

- The bounded-buffer problem was introduced in Section 6.1, it is commonly used to illustrate the power of synchronization primitives.
- Here, we present a <u>general structure of a solution</u> based on semaphore.
- In this solution, the producer and consumer processes <u>share the following data structures</u>:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The Bounded-Buffer Problem

- The code for the producer process is shown below:

```
while (true) {
        . . .
    /* produce an item in next_produced */
        . . .
    wait(empty);
    wait(mutex);
        . . .
    /* add next_produced to the buffer */
        . . .
    signal(mutex);
    signal(full);
}
```

The Bounded-Buffer Problem

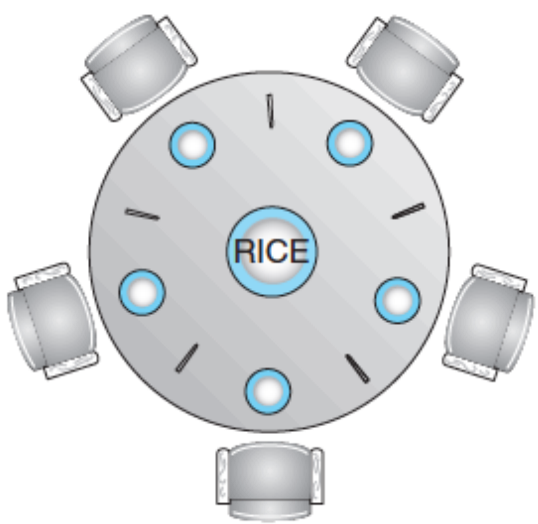
- The code for the consumer process is shown below:

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

- Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

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.
  - A bowl of rice is at the center of the table, and there are five single chopsticks on the table.
  - When a philosopher thinks, the philosopher does not interact with their colleagues.



The Dining-Philosophers Problem

- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to them (the chopsticks that are between the philosopher and their left/right neighbors).
- A philosopher may pick up only one chopstick at a time.
- Obviously, a philosopher cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both chopsticks at the same time, the philosopher eats without releasing the chopsticks.
- When a philosopher is finished eating, the philosopher puts down both chopsticks and starts thinking again.

The Dining-Philosophers Problem

- One simple solution is to represent each chopstick with a semaphore.
  - The shared data is "semaphore chopstick[5]", where all the elements of chopstick are initialized to 1.
  - A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
  - The philosopher releases both chopsticks by executing the signal() operation on the appropriate semaphores.

The Dining-Philosophers Problem

- The structure of philosopher *i* is shown below:

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

The Dining-Philosophers Problem

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

  - Suppose that all five philosophers become hungry at the same time and each grabs the left chopstick.

  - All the elements of chopstick will now be equal to 0.

  - When each philosopher tries to grab the right chopstick, they will be delayed forever.

The Dining-Philosophers Problem

- Several possible remedies to the deadlock problem are the following:

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

POSIX Mutex Locks

- In this section, we cover mutex locks and semaphores that are available in the Pthreads and POSIX APIs.

- These APIs are widely used for thread creation and synchronization by developers on UNIX, Linux, and macOS systems.

- Mutex lock represents the fundamental synchronization technique used with Pthreads.
  - A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section.

POSIX Mutex Locks

- Pthreads uses the pthread_mutex_t data type for mutex locks.
- A mutex is created with the pthread_mutex_init() function.
- The first parameter is a pointer to the mutex.
- By passing NULL as a second parameter, we initialize the mutex with its default attributes.

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

POSIX Mutex Locks

- The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions.
- If the mutex lock is unavailable, when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock().
- The following code illustrates how to protect a critical section with mutex locks:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Mutex Locks

- When a mutex is not used any more, pthread_mutex_destroy() could be used to eliminate the mutex.

  pthread_mutex_destroy(&mutex)

- All pthread mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.
- To compile a program involving pthread mutex lock, you can use the following compiling command:

```
gcc filename.c -lpthread
```

- Here is one example illustrating how mutex lock is used:

  https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

“l” is the lower case L and it stands for link

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter = 0;

void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0x4FFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

Without mutex lock, here is the output:

```
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished
```

Nothing is passed to the created thread.

POSIX Mutex Locks

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter = 0;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for (i = 0; i < (0x4FFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

POSIX Mutex Locks

```
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    pthread_mutex_destroy(&lock);

    return 0;
}
```

With mutex lock, here is the output:

```
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

POSIX Semaphores

- Many systems that implement Pthreads also provide semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension.

- POSIX specifies <u>two types of semaphores</u>: named and unnamed.

    - Fundamentally, these two types of semaphores are <u>quite similar</u>, but they <u>differ in terms of how they are created and shared</u> between processes.

    - Beginning with Version 2.6 of the kernel, Linux provides support for both named and unnamed semaphores.

POSIX Named Semaphores

- The function sem_open() is used to create and open a POSIX named semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- In this example:
  - We are naming the semaphore SEM.
  - The O_CREAT flag indicates that the semaphore will be created if it does not already exist.
  - Additionally, the semaphore can be accessed via read and write (via the parameter 0666) and is initialized to 1.

POSIX Named Semaphores

- The advantage of named semaphores is that multiple unrelated processes can easily use a common semaphore as a synchronization mechanism by simply referring to the semaphore's name.

- In the example above, once the semaphore SEM has been created, subsequent calls to sem_open() with the same parameters by other processes return a descriptor to the existing semaphore.

- Previously, we described the classic wait() and signal() semaphore operations. In POSIX, they are implemented as sem_wait() and sem_post(), respectively.

POSIX Named Semaphores

- The following code sample illustrates how to protect a critical section using the named semaphore created previously:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- An unnamed semaphore is created and initialized using the sem_init() function, which involves three parameters:
  - A pointer to the semaphore
  - A flag indicating the level of sharing
  - The semaphore's initial value

- It is illustrated in the following programming example:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

POSIX Unnamed Semaphores

- In this example, by passing the flag 0, we are indicating that this semaphore can be shared only by threads belonging to the process that created the semaphore.
  - If we supplied a nonzero value, we could allow the semaphore to be shared with separate processes by placing it in a region of shared memory.

- In addition, we initialize the semaphore to the value 1 in this example.

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

POSIX Unnamed Semaphores

- POSIX unnamed semaphores use the same sem_wait() and sem_post() operations as named semaphores.

- The following code sample illustrates how to protect a critical section using the unnamed semaphore created above:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Unnamed Semaphores

- POSIX unnamed semaphores use sem_destroy() to eliminate an unamed semaphore.

```
sem_destroy(&sem)
```

- Just like mutex locks, all semaphore functions return 0 when successful and nonzero when an error condition occurs.
- When you compile a program involving POSIX semaphore, you need to use the following command:

```
gcc filename.c -lpthread -lrt
```

  - Note that "`-lrt`" is not required for glibc 2.17 and later versions because the real time library becomes part of glibc.
  - To check the glibc version installed on Linux, run the command: `ldd --version`
- Unnamed semaphore works on Linux, but it does not work on macOS.

POSIX Unnamed Semaphores

- An example illustrating how to use unnamed semaphores can be found on the following slide.

- The details of the example are available here:

  https://www.geeksforgeeks.org/use-posix-semaphores-c/

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

void* thread(void* arg)
{
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    printf("\nJust Exiting...\n");
    return 0;
}

int main()
{
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

Without semaphore, here is the output:

```
Entered..


Entered..


Just Exiting...


Just Exiting...
```

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
    return 0;
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

The name of the semaphore is “mutex” because binary semaphore is similar to mutex lock.

With semaphore, here is the output:

```
Entered..

Just Exiting...

Entered..

Just Exiting...
```