Producer–consumer synchronization and semaphores

Although busy-waiting is generally wasteful of CPU resources, it does have the property that we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on.

- With mutexes, the order in which the threads execute the critical section is left to chance and the system.
- Since addition is commutative, this doesn't matter in our program for estimating π.
- However, it's not difficult to think of situations in which we also want to control the order in which the threads execute the code in the critical section.
  - For example, suppose each thread generates an n × n matrix, and we want to multiply the matrices together in thread-rank order. Since matrix multiplication isn't commutative, our mutex solution would have problems:

```
/* n and product_matrix are shared and initialized by
 * the main thread. product_matrix is initialized
 * to be the identity matrix. */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

    - Here we see that we wont know which threads will access the crit section (shared matrix) in what order thus we wont get a matrix back that is in thread order 0,1,2,3,..
  - A somewhat more complicated example involves having each thread "send a message" to another thread.
    - For example, suppose we have thread_count or t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, . . . , thread t − 2 to send a message to thread t − 1 and thread t − 1 to send a message to thread 0.
    - After a thread "receives" a message, it can print the message and terminate.
    - To implement the message transfer, we can allocate a shared array of char∗. Then each thread can allocate storage for the message it's sending, and, after it has initialized the message, set a pointer in the shared array to refer to it.
      - To avoid dereferencing undefined pointers, the main thread can set the individual entries in the shared array to NULL.

```
1  /* 'messages' has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main.                  */
3  void *Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     long source = (my_rank + thread_count - 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11
12    if (messages[my_rank] != NULL)
13       printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14    else
15       printf("Thread %ld > No message from %ld\n",
16              my_rank, source);
17
18    return NULL;
19 }  /* Send_msg */
```

Program 4.7: A first attempt at sending messages using pthreads.

Note:
- Source is the overall issue.
- Source is the previous thread to the current thread we are on
- Line 14 says that if we saw null then the source didn't give us a message

      - When we run the program with more than a couple of threads on a dual core system, we see that some of the messages are never received.

- For example, thread 0, which is started first, will typically finish before thread t − 1 has copied the message into the messages array
- This isn't surprising, and we could fix the problem by replacing the if statement in Line 12 with a busy-wait while statement:
- ```
  while (messages[my_rank] == NULL);
  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
  ```
  - This will wait for the thread to have received a message before printing
- Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach.

- A different approach:mutex(but not so good):

After executing the assignment in Line 10, we'd like to "notify" the thread with rank `dest` that it can proceed to print the message. We'd like to do something like this:

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

- The current thread has added its msg into the messages array and now the next thread can add its own msg
- ```
  1     . . .
  2     pthread_mutex_lock(&mutex[dest]);
  3     . . .
  4     messages[dest] = my_msg;
  5     pthread_mutex_unlock(&mutex[dest]);
  6     . . .
  7     pthread_mutex_lock(&mutex[my_rank]);
  8     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
  9     . . .
  ```

"It's not at all clear how mutexes can help here. We might try calling pthread_mutex_unlock to "notify" the thread dest. However, mutexes are initialized to be unlocked, so we'd need to add a call before initializing messages[dest] to lock the mutex. This will be a problem, since we don't know when the threads will reach the calls to pthread_mutex_lock."

- This doesn't work because say we have two threads. thread 0 gets so far ahead of thread 1 that it reaches the second call to pthread_mutex_lock in Line 7 before thread 1 reaches the first in Line 2. Then, of course, it will acquire the lock and continue to the printf statement. This will result in thread 0 dereferencing a null pointer and it will crash.
- Essentially, thread 0 doesn't get locked in time (on line 2) by thread 1 because thread 0 is so far ahead. This causes thread 0 to execute line 8 which tries to do a printf statement that will try to dereference messages[my_rank] which will be null since thread 1 hasn't put in its own message yet.

A solution is called **Semaphore**:

- A semaphore can be thought of as a special type of unsigned int, so they take on the values 0, 1, 2, ... In many cases, we'll only be interested in using them when they take on the values 0 and 1.
  - A semaphore that only takes on these values is called a binary semaphore
- Very roughly speaking, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex.
- Summary of how it operates:
  - "Before the critical section you want to protect, you place a call to the function sem_wait. A thread that executes sem_wait will block if the semaphore is 0. If the semaphore is nonzero, it will decrement the semaphore and proceed. After executing the code in the critical section, a thread calls sem_post, which increments the semaphore, and a thread waiting in sem_wait can proceed."
- For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore. The main thread can initialize all of the semaphores to 0—that is,

"locked"—and then any thread can execute a sem_post on any of the semaphores.
  - Similarly, any thread can execute sem_wait on any of the semaphores.
- Semaphor function:

The syntax of the various semaphore functions is

```
int sem_init(
      sem_t*     semaphore_p    /* out */,
      int        shared         /* in  */,
      unsigned   initial_val    /* in  */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

  - The second argument to sem_init controls whether the semaphore is shared among threads or processes. In our examples, we'll be sharing the semaphore among threads, so the constant 0 can be passed in.
  - Note that semaphores are part of the POSIX standard, but not part of Pthreads. Hence it is necessary to ensure your operating system does indeed support semaphores, and then add the following preprocessor directive to any program that uses them

```
#include <semaphore.h>
```

- Thus, if we use semaphores, our Send_msg function can be written as

```
1  /* 'messages' is allocated and initialized to NULL in main */
2  /* 'semaphores' is allocated and initialized to            */
3  /* 0 (locked) in main                                      */
4  void *Send_msg(void* rank) {
5     long my_rank = (long) rank;
6     long dest = (my_rank + 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11    /* ''Unlock'' the semaphore of dest: */
12    sem_post(&semaphores[dest]);
13
14    /* Wait for our semaphore to be unlocked */
15    sem_wait(&semaphores[my_rank]);
16    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
17
18    return NULL;
19 } /* Send_msg */
```

Program 4.8: Using semaphores so that threads can send messages.

- Finally, note that the message-sending problem didn't involve a critical section. The problem wasn't that there was a block of code that could only be executed by one thread at a time. Rather, thread my_rank couldn't proceed until thread source had finished creating the message.
  - This type of synchronization, when a thread can't proceed until another thread has taken some action, is sometimes called producer–consumer synchronization.
- **Counting semaphores** can also be useful in scenarios where we wish to restrict access to a finite resource.

ios where we wish to restrict access to a finite resource. One common example is an application design pattern that involves limiting the number of threads used by a program to be no more than the number of cores available on a given machine. Consider a program with a workload of N tasks, where N is much greater than the available cores. In this case, the main thread is responsible for distributing the workload and would initialize its semaphore with the number of cores available, and then call `sem_wait` before starting each worker thread with `pthread_create`. Once the counter reaches 0, the main thread will block; the machine has a task running for each core and the program must wait for a thread to finish before starting more. When a thread does finish its task, it will call `sem_post` to signal that the main thread can create another worker thread. For this approach to be efficient, the amount of time spent on each task much be longer than the thread creation overhead because N total threads will be started during the program's execution. For an approach that reuses existing threads in a *thread pool*, see Programming Assignment 4.5.