4 Concurrent Programming

Abstract
This chapter covers concurrent programming. It introduces the concept of parallel computing and points out its importance. It compares sequential algorithms with parallel algorithms, and parallelism vs. concurrency. It explains the principles of threads and their advantages over processes. It covers threads operations in Pthreads by examples. These include threads management functions, threads synchronization tools of mutex, join, condition variables and barriers. It demonstrates concurrent programming using threads by detailed examples. These include matrix computation, quicksort and solving systems of linear equations by concurrent threads. It explains the deadlock problem and shows how to prevent deadlocks in concurrent programs. It covers semaphores and demonstrates their advantages over condition variables. It also explains the unique way of supporting threads in Linux. The programming project is to implement user-level threads. It presents a base system to help the reader get started. The base system supports dynamic creation, execution and termination of concurrent tasks, which are equivalent to threads executing in the same address space of a process. The project is for the reader to implement threads join, mutex and semaphores for threads synchronization and demonstrate their usage in concurrent programs. The programming project should allow the reader to have a deeper understanding of the principles and techniques of multitasking, threads synchronization and concurrent programming.

4.1 Introduction to Parallel Computing

In the early days, most computers have only one processing element, known as the processor or Central Processing Unit (CPU). Due to this hardware limitation, computer programs are traditionally written for serial computation. To solve a problem, a person would first design an algorithm, which describes how to solve the problem step by step, and then implement the algorithm by a computer program as a serial stream of instructions. With only a single CPU, both the individual instructions and the steps of an algorithm can only be executed sequentially one at a time. However, algorithms based on the principle of divide and conquer, e.g. binary search and quicksort, etc. often exhibit a high degree of parallelism, which can be exploited by using parallel or concurrent executions to speed up the computation. Parallel computing is a computing scheme which tries to use multiple processors executing parallel algorithms to solve problems faster. In the past, parallel computing is rarely

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978 3 319 92429 8 4

accessible to average programmers due to its extensive requirements of computing resources. With the advent of multicore processors in recent years, most operating systems, such as Linux, support Symmetrical Multiprocessing (SMP). Parallel computing has become a reality even for average programmers. The future of computing is clearly in the direction of parallel computing. It is imperative to introduce parallel computing to Computer Science and Computer Engineering students in an early stage. In this chapter, we shall cover the basic concepts and techniques of parallel computing through concurrent programming.

4.1.1 Sequential Algorithms vs. Parallel Algorithms

When describing sequential algorithms, a common way is to express the algorithm in a **begin-end** block, as show in the left-hand side of the following diagram.

```
       Sequential Algorithm   |    Parallel Algorithm
          begin               |      cobegin
            step 1            |        task 1
            step 2            |        task 2
            . . .             |        . . .
            step n            |        task n
          end                 |      coend
          // next step        |      // next step
    ----------------------------------------------------
```

The sequential algorithm inside the begin-end block may consist of many steps. All the steps are to be performed by a single task serially one step at a time. The algorithm ends when all the steps have completed. In contrast, the right-hand side of the diagram shows the description of a parallel algorithm, which uses a **cobegin-coend** block to specify separate tasks of a parallel algorithm. Within the cobegin-coend block, all tasks are to be performed in parallel. The next step following the cobegin-coend block will be performed only after all such tasks have completed.

4.1.2 Parallelism vs. Concurrency

In general, a parallel algorithm only identifies tasks that can be executed in parallel, but it does not specify how to map the tasks to processing elements. Ideally, all the tasks in a parallel algorithm should be executed simultaneously in real time. However, true parallel executions can only be achieved in systems with multiple processing elements, such as multiprocessor or multicore systems. On single CPU systems, only one task can execute at a time. In this case, different tasks can only execute concurrently, i.e. logically in parallel. In single CPU systems, concurrency is achieved through multitasking, which was covered in Chap. 3. We shall explain and demonstrate the principle and techniques of multitasking again in a programming project at the end of this chapter.

4.2 Threads

4.2.1 Principle of Threads

An operating system (OS) comprises many concurrent processes. In the process model, processes are independent execution units. Each process executes in either kernel mode or user mode. While in user mode, each process executes in a unique address space, which is separated from other processes. Although each process is an independent unit, it has only one execution path. Whenever a process must wait for something, e.g. an I/O completion event, it becomes suspended and the entire process execution stops. Threads are independent execution units in the same address space of a process. When creating a process, it is created in a unique address space with a main thread. When a process begins, it executes the main thread of the process. With only a main thread, there is virtually no difference between a process and a thread. However, the main thread may create other threads. Each thread may create yet more threads, etc. All threads in a process execute in the same address space of the process but each thread is an independent execution unit. In the threads model, if a thread becomes suspended, other threads may continue to execute. In addition to sharing a common address space, threads also share many other resources of a process, such as user id, opened file descriptors and signals, etc. A simple analogy is that a process is a house with a house master (the main thread). Threads are people living in the same house of a process. Each person in a house can carry on his/her activities independently, but they share some common facilities, such as the same mailbox, kitchen and bathroom, etc. Historically, most computer vendors used to support threads in their own proprietary operating systems. The implementations vary considerably from system to system. Currently, almost all OS support Pthreads, which is the threads standard of IEEE POSIX 1003.1c (POSIX 1995). For more information, the reader may consult numerous books (Buttlar et al. 1996) and online articles on Pthreads programming (Pthreads 2017).

4.2.2 Advantages of Threads

Threads have many advantages over processes.

(1). **Thread creation and switching are faster:** The context of a process is complex and large. The complexity stems mainly from the need for managing the process image. For example, in a system with virtual memory, a process image may be composed of many units of memory called **pages.** During execution some of the pages are in memory while others may not. The OS kernel must use several page tables and many levels of hardware assistances to keep track of the pages of each process. To create a new process, the OS must allocate memory and build the page tables for the process. To create a thread within a process, the OS does not have to allocate memory and build page tables for the new thread since it shares the same address space of the process. So, creating a thread is faster than creating a process. Also, thread switching is faster than process switching for the following reasons. Process switching involves replacing the complex paging environment of one process with that of another, which requires a lot of operations and time. In contrast, switching among threads in the same process is much simpler and faster because the OS kernel only needs to switch the execution points without changing the process image.

(2). **Threads are more responsive:** A process has only a single execution path. When a process becomes suspended, the entire process execution stops. In contrast, when a thread becomes suspended, other threads in the same process can continue to execute. This allows a program with threads to be more responsive. For example, in a process with multiple threads, while one thread becomes blocked to wait for I/O, other threads can still do computations in the background. In a server with threads, the server can serve multiple clients concurrently.

(3). **Threads are better suited to parallel computing:** The goal of parallel computing is to use multiple execution paths to solve problems faster. Algorithms based on the principle of divide and conquer, e.g. binary search and quicksort, etc. often exhibit a high degree of parallelism, which can be exploited by using parallel or concurrent executions to speed up the computation. Such algorithms often require the execution entities to share common data. In the process model, processes cannot share data efficiently because their address spaces are all distinct. To remedy this problem, processes must use **Interprocess Communication (IPC)** to exchange data or some other means to include a common data area in their address spaces. In contrast, threads in the same process share all the (global) data in the same address space. Thus, writing programs for parallel executions using threads is simpler and more natural than using processes.

4.2.3 Disadvantages of Threads

On the other hand, threads also have some disadvantages, which include

(1). Because of shared address space, threads needs explicit synchronization from the user.

(2). Many library functions may not be threads safe, e.g. the traditional strtok() function, which divides a string into tokens in-line. In general, any function which uses global variables or relies on contents of static memory is not threads safe. Considerable efforts are needed to adapt library functions to the threads environment.

(3). On single CPU systems, using threads to solve problems is actually slower than using a sequential program due to the overhead in threads creation and context switching at run-time.

4.3 Threads Operations

The execution locus of a thread is similar to that a process. A thread can execute in either kernel mode or user mode. While in user mode, threads executes in the same address space of a process but each has its own execution stack. As an independent execution unit, a thread can make system calls to the OS kernel, subject to the kernel's scheduling policy, becomes suspended, activated to resume execution, etc. To take advantage of the shared address space of threads, the OS kernel's scheduling policy may favor threads of the same process over those in different processes. As of now, almost all operating systems support POSIX Pthreads, which defines a standard set of Application Programming Interfaces (**APIs**) to support threads programming. In the following, we shall discuss and demonstrate concurrent programming by Pthreads in Linux (Goldt et al 1995; IBM; Love 2005; Linux Man Page Project 2017).

4.4 Threads Management Functions

The Pthreads library offers the following APIs for threads management.

```
pthread create(thread, attr, function, arg): create thread
pthread exit(status)       : terminate thread
```

```
    pthread cancel(thread)     : cancel thread
    pthread attr init(attr)    : initialize thread attributes
    pthread attr destroy(attr): destroy thread attribute
```

4.4.1 Create Thread

threads are created by the pthread create() function.

```
          int pthread create (pthread t *pthread id, pthread attr t *attr,
                              void *(*func)(void *),  void *arg);
```

which returns 0 on success or an error number on failure. Parameters to the pthread create() function are

.pthread id is a pointer to a variable of the pthread t type. It will be filled with the unique thread ID assigned by the OS kernel. In POSIX, pthread t is an opaque type. The programmer should not know the contents of an opaque object because it may depend on implementation. A thread may get its own ID by the pthread self() function. In Linux, pthread t type is defined as **unsigned long**, so thread ID can be printed as %lu.
.attr is a pointer to another opaque data type, which specifies the thread attributes, which are explained in more detail below.
.func is the entry address of a function for the new thread to execute.
.arg is a pointer to a parameter for the thread function, which can be written as

```
              void *func(void *arg)
```

Among these, the attr parameter is the most complex one. The steps of using an attr parameter are as follows.

(1). Define a pthread attribute variable `pthread attr t` **attr**
(2). Initialize the attribute variable with `pthread attr init(&attr)`
(3). Set the attribute variable and use it in pthread create() call
(4). If desired, free the attr resource by `pthread attr destroy(&attr)`

The following shows some examples of using the attribute parameter. By default, every thread is created to be joinable with other threads. If desired, a thread can be created with the detached attribute, which makes it non-joinable with other threads. The following code segment shows how to create a detached thread.

```
   pthread attr t attr;             // define an attr variable
   pthread attr init(&attr);        // initialize attr
   pthread attr setdetachstate(&attr, PTHREAD CREATE DETACHED); // set attr
   pthread create(&thread id, &attr, func, NULL);  // create thread with attr
   pthread attr destroy(&attr);     // optional: destroy attr
```

Every thread is created with a default stack size. During execution, a thread may find out its stack size by the function

```
            size t pthread attr getstacksize()
```

which returns the default stack size. The following code segment shows how to create a thread with a specific stack size.

```
pthread attr t attr;           // attr variable
size t stacksize;              // stack size
pthread attr init(&attr);      // initialize attr
stacksize = 0x10000;           // stacksize=16KB;
pthread attr setstacksize (&attr, stacksize);   // set stack size in attr
pthread create(&threads[t], &attr, func, NULL); // create thread with stack
                                                   size
```

If the attr parameter is NULL, threads will be created with default attributes. In fact, this is the recommended way of creating threads, which should be followed unless there is a compelling reason to alter the thread attributes. In the following, we shall always use the default attributes by setting attr to NULL.

4.4.2 Thread ID

Thread ID is an opaque data type, which depends on implementation. Therefore, thread IDs should not be compared directly. If needed, they can be compared by the `pthread equal() function.`

```
int pthread equal (pthread t t1, pthread t t2);
```

which returns zero if the threads are different threads, non-zero otherwise.

4.4.3 Thread Termination

A thread terminates when the thread function finishes. Alternatively, a thread may call the function

```
int pthread exit (void *status);
```

to terminate explicitly, where status is the exit status of the thread. As usual, a 0 exit value means normal termination, and non-zero values mean abnormal termination.

4.4.4 Thread Join

A thread can wait for the termination of another thread by

```
int pthread join (pthread t thread, void **status ptr);
```

The exit status of the terminated thread is returned in status ptr.

4.5 Threads Example Programs

4.5.1 Sum of Matrix by Threads

Example 4.1: Assume that we want to compute the sum of all the elements in an N × N matrix of integers. The problem can be solved by a concurrent algorithm using threads. In this example, the main thread first generates an N × N matrix of integers. Then it creates N working threads, passing as parameter a unique row number to each working thread, and waits for all the working threads to terminate. Each working thread computes the partial sum of a distinct row, and deposits the partial sum in a corresponding row of a global array int sum[N]. When all the working threads have finished, the main thread resumes. It computes the total sum by adding the partial sums generated by the working threads. The following shows the complete C code of the example program C4.1. Under Linux, the program must be compiled as

```
gcc  C4.1.c  -pthread
```

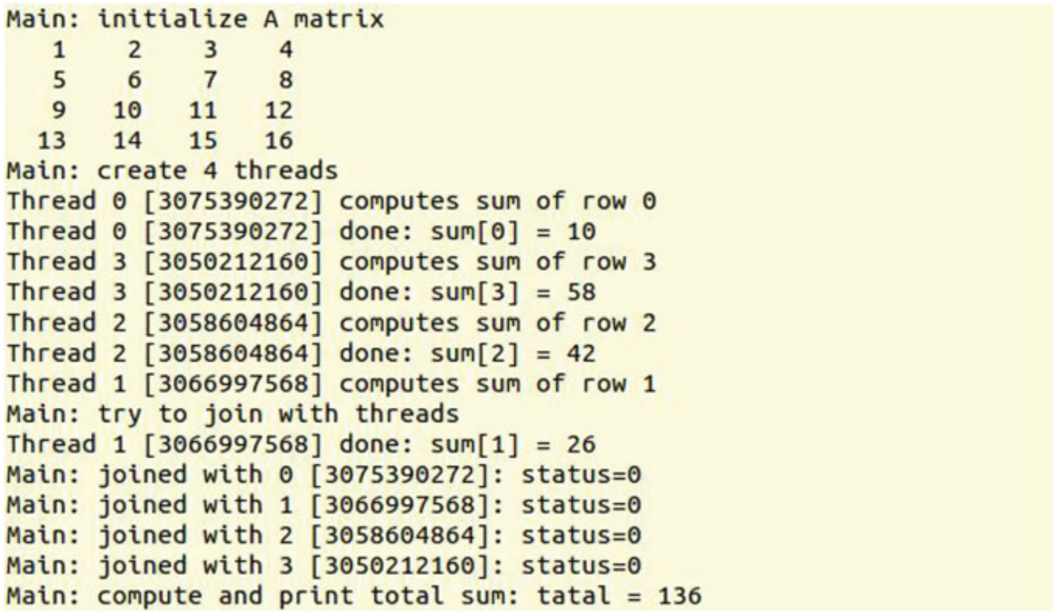
```
/**** C4.1.c file: compute matrix sum by threads ***/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define  N   4
int A[N][N], sum[N];

void *func(void *arg)              // threads function
{
   int j, row;
   pthread t tid = pthread self(); // get thread ID number
   row = (int)arg;                 // get row number from arg
   printf("Thread %d [%lu] computes sum of row %d\n", row, tid, row);
   for (j=0; j<N; j++)      // compute sum of A[row]in global sum[row]
       sum[row] += A[row][j];
   printf("Thread %d [%lu] done: sum[%d] = %d\n",
                  row, tid, row, sum[row]);
   pthread exit((void*)0); // thread exit: 0=normal termination
}

int main (int argc, char *argv[])
{
   pthread t thread[N];      // thread IDs
   int i, j, r, total = 0;
   void *status;
   printf("Main: initialize A matrix\n");
   for (i=0; i<N; i++){
     sum[i] = 0;
     for (j=0; j<N; j++){
       A[i][j] = i*N + j + 1;
       printf("%4d ", A[i][j]);
     }
     printf("\n");
   }
   printf("Main: create %d threads\n", N);
   for(i=0; i<N; i++) {
      pthread create(&thread[i], NULL, func, (void *)i);
   }
   printf("Main: try to join with threads\n");
   for(i=0; i<N; i++) {
      pthread join(thread[i], &status);
      printf("Main: joined with %d [%lu]: status=%d\n",
                               i, thread[i], (int)status);
   }
   printf("Main: compute and print total sum: ");
   for (i=0; i<N; i++)
       total += sum[i];
   printf("tatal = %d\n", total);
   pthread exit(NULL);
}
```

Figure 4.1 shows the outputs of running the Example Program C4.1. It shows the executions of individual threads and their computed partial sums. It also demonstrates the thread join operation.

4.5.2 Quicksort by Threads

Example 4.2: Quicksort by Concurrent Threads

In this example, we shall implement a parallel quicksort program by threads. When the program starts, it runs as the main thread of a process. The main thread calls qsort(&arg), with arg =[lowerbound=0, upperbound=N-1]. The qsort() function implements quicksort of an array of N integers. In qsort(), the thread picks a pivot element to divide the array into two parts such that all elements in the left part are less than the pivot and all elements in the right part are greater than the



```
Main: initialize A matrix
   1    2    3    4
   5    6    7    8
   9   10   11   12
  13   14   15   16
Main: create 4 threads
Thread 0 [3075390272] computes sum of row 0
Thread 0 [3075390272] done: sum[0] = 10
Thread 3 [3050212160] computes sum of row 3
Thread 3 [3050212160] done: sum[3] = 58
Thread 2 [3058604864] computes sum of row 2
Thread 2 [3058604864] done: sum[2] = 42
Thread 1 [3066997568] computes sum of row 1
Main: try to join with threads
Thread 1 [3066997568] done: sum[1] = 26
Main: joined with 0 [3075390272]: status=0
Main: joined with 1 [3066997568]: status=0
Main: joined with 2 [3058604864]: status=0
Main: joined with 3 [3050212160]: status=0
Main: compute and print total sum: tatal = 136
```

Fig. 4.1 Sample Outputs of Example 4.1

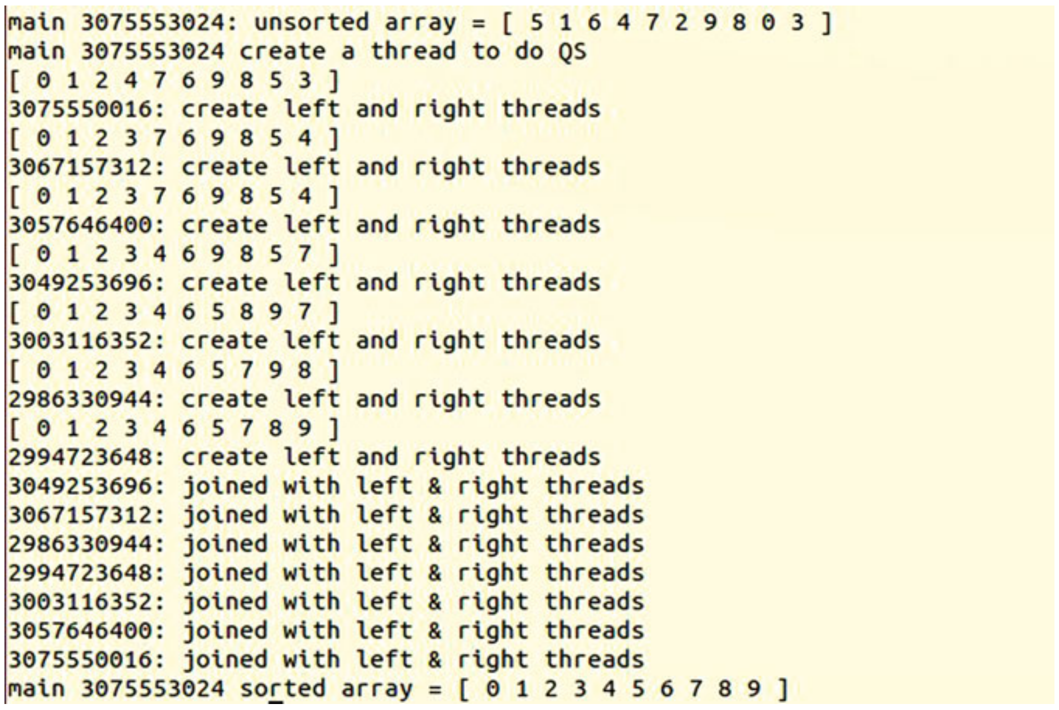
pivot. Then it creates two subthreads to sort each of the two parts, and waits for the subthreads to finish. Each subthread sorts its own range by the same algorithm recursively. When all the subthreads have finished, the main thread resumes. It prints the sorted array and terminates. As is well known, the number of sorting steps of quicksort depends on the order of the unsorted data, which affects the number of threads needed in the qsort program.

```
/****** C4.2.c: quicksort by threads *****/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct{
  int upperbound;
  int lowerbound;
}PARM;

#define N 10
int A[N] = {5,1,6,4,7,2,9,8,0,3};  // unsorted data

int print()   // print current a[] contents
{
  int i;
  printf("[ ");
  for (i=0; i<N; i++)
    printf("%d ", a[i]);
  printf("]\n");
}

void *qsort(void *aptr)
{
  PARM *ap, aleft, aright;
  int pivot, pivotIndex, left, right, temp;
  int upperbound, lowerbound;

  pthread t me, leftThread, rightThread;
  me = pthread self();
  ap = (PARM *)aptr;
  upperbound = ap >upperbound;
  lowerbound = ap >lowerbound;
  pivot = a[upperbound];         // pick low pivot value
  left = lowerbound   1;         // scan index from left side
  right = upperbound;            // scan index from right side
  if (lowerbound >= upperbound)
     pthread exit(NULL);

  while (left < right) {         // partition loop
    do { left++;} while (a[left] < pivot);
       do { right  ;} while (a[right] > pivot);
       if (left < right ) {
          temp = a[left];
          a[left] = a[right];
          a[right] = temp;
       }
    }
    print();
    pivotIndex = left;           // put pivot back
    temp = a[pivotIndex];
    a[pivotIndex] = pivot;
    a[upperbound] = temp;
    // start the "recursive threads"
    aleft.upperbound = pivotIndex   1;
    aleft.lowerbound = lowerbound;
    aright.upperbound = upperbound;
    aright.lowerbound = pivotIndex + 1;
    printf("%lu: create left and right threads\n", me);
    pthread create(&leftThread,  NULL, qsort, (void *)&aleft);
    pthread create(&rightThread, NULL, qsort, (void *)&aright);
    // wait for left and right threads to finish
    pthread join(leftThread, NULL);
    pthread join(rightThread, NULL);
    printf("%lu: joined with left & right threads\n", me);
}
```

`int main(int argc, char *argv[])`

```
{
    PARM arg;
    int i, *array;
    pthread t me, thread;
    me = pthread self();
    printf("main %lu: unsorted array = ", me);
    print();
    arg.upperbound = N 1;
    arg.lowerbound = 0;
    printf("main %lu create a thread to do QS\n", me);
    pthread create(&thread, NULL, qsort, (void *)&arg);
    // wait for QS thread to finish
    pthread join(thread, NULL);
    printf("main %lu sorted array = ", me);
    print();
}
```

Figure 4.2 shows the outputs of running the Example Program C4.2, which demonstrates parallel quicksort by concurrent threads.



```
main 3075553024: unsorted array = [ 5 1 6 4 7 2 9 8 0 3 ]
main 3075553024 create a thread to do QS
[ 0 1 2 4 7 6 9 8 5 3 ]
3075550016: create left and right threads
[ 0 1 2 3 7 6 9 8 5 4 ]
3067157312: create left and right threads
[ 0 1 2 3 7 6 9 8 5 4 ]
3057646400: create left and right threads
[ 0 1 2 3 4 6 9 8 5 7 ]
3049253696: create left and right threads
[ 0 1 2 3 4 6 5 8 9 7 ]
3003116352: create left and right threads
[ 0 1 2 3 4 6 5 7 9 8 ]
2986330944: create left and right threads
[ 0 1 2 3 4 6 5 7 8 9 ]
2994723648: create left and right threads
3049253696: joined with left & right threads
3067157312: joined with left & right threads
2986330944: joined with left & right threads
2994723648: joined with left & right threads
3003116352: joined with left & right threads
3057646400: joined with left & right threads
3075550016: joined with left & right threads
main 3075553024 sorted array = [ 0 1 2 3 4 5 6 7 8 9 ]
```

Fig. 4.2 Outputs of Parallel Quicksort Program

4.6 Threads Synchronization

Since threads execute in the same address space of a process, they share all the global variables and data structures in the same address space. When several threads try to modify the same shared variable or data structure, if the outcome depends on the execution order of the threads, it is called a **race condition**. In concurrent programs, race conditions must not exist. Otherwise, the results may be inconsistent. In addition to the join operation, concurrently executing threads often need to cooperate with one another. In order to prevent race conditions, as well as to support threads cooperation, threads need synchronization. In general, synchronization refers to the mechanisms and rules used to ensure the integrity of shared data objects and coordination of concurrently executing entities. It can be applied to either processes in kernel mode or threads in user mode. In the following, we shall discuss the specific problem of threads synchronization in Pthreads.

4.6.1 Mutex Locks

The simplest kind of synchronizing tool is a lock, which allows an execution entity to proceed only if it possesses the lock. In Pthreads, locks are called **mutex,** which stands for **Mutual Exclusion**. Mutex variables are declared with the type pthread mutex t, and they must be initialized before using. There are two ways to initialize a mutex variable.

(1) Statically, as in

```
pthread mutex t m =  PTHREAD MUTEX INITIALIZER;
```

which defines a mutex variable m and initializes it with default attributes.

(2) Dynamically with the **pthread mutex init()** function, which allows setting the mutex attributes by an attr parameter, as in

```
pthread mutex init (pthread mutex t *m, pthread mutexattr t,*attr);
```

As usual, the attr parameter can be set to NULL for default attributes.

After initialization, mutex variables can be used by threads via the following functions.

```
int pthread mutex lock (pthread mutex t *m);     // lock mutex
int pthread mutex unlock (pthread mutex t *m);   // unlock mutex
int pthread mutex trylock (pthread mutex t *m);  // try to lock mutex
int pthread mutex destroy (pthread mutex t *m);  // destroy mutex
```

Threads use mutexes as locks to protect shared data objects. A typical usage of mutex is as follows. A thread first creates a mutex and initializes it once. A newly created mutex is in the unlocked state and without an owner. Each thread tries to access a shared data object by

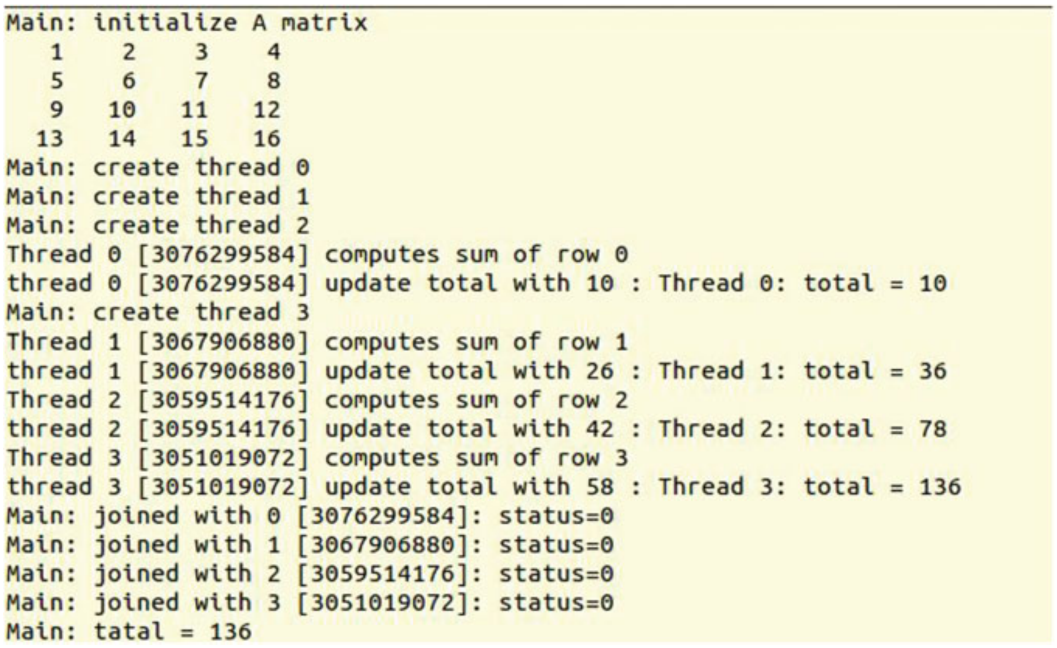
```
pthread mutex lock(&m);    // lock mutex
 access shared data object; // access shared data in a critical region
pthread mutex unlock(&m);  // unlock mutex
```

When a thread executes pthread mutex lock(&m), if the mutex is unlocked, it locks the mutex, becomes the mutex owner and continues. Otherwise, it becomes blocked and waits in the mutex waiting queue. Only the thread which has acquired the mutex lock can access the shared data object. A sequence of executions which can only be performed by one execution entity at a time is commonly known as a **Critical Region** (CR). In Pthreads, mutexes are used as locks to protect Critical Regions to ensure at most one thread can be inside a CR at any time. When the thread finishes with the shared data object, it exits the CR by calling pthread mutex unlock(&m) to unlock the mutex. A locked mutex can only be unlocked by the current owner. When unlocking a mutex, if there are no blocked threads in the mutex waiting queue, it unlocks the mutex and the mutex has no owner. Otherwise, it unblocks a waiting thread from the mutex waiting queue, which becomes the new owner, and the mutex remains locked. When all the threads are finished, the mutex may be destroyed if it was dynamically allocated. We demonstrate threads synchronization using mutex lock by an example.

Example 4.3: This example is a modified version of Example 4.1. As before, we shall use N working threads to compute the sum of all the elements of an N $\times$ N matrix of integers. Each working thread computes the partial sum of a row. Instead of depositing the partial sums in a global sum[ ] array, each working thread tries to update a global variable, total, by adding its partial sum to it. Since all the working threads try to update the same global variable, they must be synchronized to prevent race conditions. This can be achieved by a mutex lock, which ensures that only one working thread can update the total variable at a time in a Critical Region. The following shows the Example Program C4.3.

```
/** C4.3.c: matrix sum by threads with mutex lock **/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
#define  N   4
int A[N][N];

int total = 0;                 // global total
pthread mutex t *m;            // mutex pointer
void *func(void *arg)          // working thread function
{
   int i, row, sum = 0;
   pthread t tid = pthread self(); // get thread ID number
   row = (int)arg;                 // get row number from arg
   printf("Thread %d [%lu] computes sum of row %d\n", row, tid, row);
   for (i=0; i<N; i++)       // compute partial sum of A[row]in
       sum += A[row][i];
   printf("Thread %d [%lu] update total with %d : ", row, tid, sum);
   pthread mutx lock(m);
     total += sum;           // update global total inside a CR
   pthread mutex unlock(m);
   printf("total = %d\n", total);
}

int main (int argc, char *argv[])
{
   pthread t thread[N];
   int i, j, r;
   void *status;
   printf("Main: initialize A matrix\n");
   for (i=0; i<N; i++){
     sum[i] = 0;
     for (j=0; j<N; j++){
       A[i][j] = i*N + j + 1;
       printf("%4d ", A[i][j]);
     }
     printf("\n");
   }
   // create a mutex m
   m = (pthread mutex t *)malloc(sizeof(pthread mutex t));
   pthread mutex init(m, NULL); // initialize mutex m
   printf("Main: create %d threads\n", N);
   for(i=0; i<N; i++) {
      pthread create(&thread[i], NULL, func, (void *)i);
   }
   printf("Main: try to join with threads\n");
   for(i=0; i<N; i++) {
      pthread join(thread[i], &status);
      printf("Main: joined with %d [%lu]: status=%d\n",
              i, thread[i], (int)status);
   }
   printf("Main: tatal = %d\n", total);
   pthread mutex destroy(m); // destroy mutex m
   pthread exit(NULL);
}
```



```
Main: initialize A matrix
   1    2    3    4
   5    6    7    8
   9   10   11   12
  13   14   15   16
Main: create thread 0
Main: create thread 1
Main: create thread 2
Thread 0 [3076299584] computes sum of row 0
thread 0 [3076299584] update total with 10 : Thread 0: total = 10
Main: create thread 3
Thread 1 [3067906880] computes sum of row 1
thread 1 [3067906880] update total with 26 : Thread 1: total = 36
Thread 2 [3059514176] computes sum of row 2
thread 2 [3059514176] update total with 42 : Thread 2: total = 78
Thread 3 [3051019072] computes sum of row 3
thread 3 [3051019072] update total with 58 : Thread 3: total = 136
Main: joined with 0 [3076299584]: status=0
Main: joined with 1 [3067906880]: status=0
Main: joined with 2 [3059514176]: status=0
Main: joined with 3 [3051019072]: status=0
Main: tatal = 136
```

Fig. 4.3 Outputs of Example 4.3 Program

Figure 4.3 show the sample outputs of running the Example 4.3 program, which demonstrates mutex lock in Pthreads.

4.6.2 Deadlock Prevention

Mutexes use the locking protocol. If a thread can not acquire a mutex lock, it becomes blocked, waiting for the mutex to be unlocked before continuing. In any locking protocol, misuse of locks may lead to problems. The most well-known and prominent problem is deadlock. **Deadlock** is a condition, in which many execution entities mutually wait for one another so that none of them can proceed. To illustrate this, assume that a thread T1 has acquired a mutex lock m1 and tries to lock another mutex m2. Another thread T2 has acquired the mutex lock m2 and tries to lock the mutex m1, as shown in the following diagram.

```
Thread T1:          |    Thread T2:
                    |
 lock(m1);          |     lock(m2);
 . . . .            |     . . . .
 lock(m2);          |     lock(m1);
```

In this case, T1 and T2 would mutually wait for each other forever, so they are in a deadlock due to crossed locking requests. Similar to no race conditions, deadlocks must not exist in concurrent programs. There are many ways to deal with possible deadlocks, which include **deadlock prevention**, deadlock avoidance, deadlock detection and recovery, etc. In real systems, the only practical way is deadlock prevention, which tries to prevent deadlocks from occurring when designing parallel algorithms. A simple way to prevent deadlock is to order the mutexes and ensure that every thread requests mutex locks only in a single direction, so that there are no loops in the request sequences.

However, it may not be possible to design every parallel algorithm with only uni-directional locking requests. In such cases, the conditional locking function, **pthread mutex trylock(),** may be used to prevent deadlocks. The trylock() function returns immediately with an error if the mutex is already locked. In that case, the calling thread may back-off by releasing some of the locks it already holds, allowing other threads to continue. In the above crossed locking example, we may redesign one of the threads, e.g. T1, as follows, which uses conditional locking and back-off to prevent the deadlock.

```
                 Thread T1:

while(1){
   lock(m1);
   if (!trylock(m2)) // if trylock m2 fails
      unlock(m1);    // back off and retry
   else
      break;
   // delay some random time before retry
}
```

4.6.3 Condition Variables

Mutexes are used only as locks, which ensure threads to access shared data objects exclusively in Critical Regions. Condition variables provide a means for threads cooperation. Condition variables are always used in conjunction with mutex locks. This is no surprise because mutual exclusion is the basis of all synchronizing mechanisms. In Pthreads, condition variables are declared with the type **pthread cond t**, and must be initialized before using. Like mutexes, condition variables can also be initialized in two ways.

(1) Statically, when it is declared, as in

```
pthread cond t con = PTHREAD COND INITIALIZER;
```

which defines a condition variable, con, and initializes it with default attributes.

(2) Dynamically with the **pthread cond init()** function, which allows setting a condition variable with an attr parameter. For simplicity, we shall always use a NULL attr parameter for default attributes.

The following code segments show how to define a condition variable with an associated mutex lock.

```
pthread mutex t con mutex; // mutex for a condition variable
pthread cond t   con;      // a condition variable that relies on con mutex
pthread mutex init(&con mutex, NULL);  // initialize mutex
pthread cond init(&con, NULL);         // initialize con
```

When using a condition variable, a thread must acquire the associated mutex lock first. Then it performs operations within the Critical Region of the mutex lock and releases the mutex lock, as in

```
pthread mutex lock(&con mutex);
    modify or test shared data objects
    use condition variable con to wait or signal conditions
pthread mutex unlock(&con mutex);
```

Within the CR of the mutex lock, threads may use condition variables to cooperate with one another via the following functions.

`pthread cond wait(condition, mutex)`: `This function` blocks the calling thread until the specified condition is signaled. This routine should be called while mutex is locked. It will automatically release the mutex lock while the thread waits. After signal is received and a blocked thread is awakened, mutex will be automatically locked.

`pthread cond signal(condition)`: `This function` is used to signal, i.e. to wake up or unblock, a thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for **`pthread cond wait()`** to complete.

`pthread cond broadcast(condition)`: This function unblocks all threads that are blocked on the condition variable. All unblocked threads will compete for the same mutex to access the condition variable. Their order of execution depends on threads scheduling.

We demonstrate thread cooperation using condition variables by an example.

4.6.4 Producer-Consumer Problem

Example 4.4: In this example, we shall implement a simplified version of the **producer-consumer problem,** which is also known as the **bounded buffer** problem, using threads and condition variables. The producer-consumer problem is usually defined with processes as executing entities, which can be regarded as threads in the current context. The problem is defined as follows.

A set of producer and consumer processes share a finite number of buffers. Each buffer contains a unique item at a time. Initially, all the buffers are empty. When a producer puts an item into an empty buffer, the buffer becomes full. When a consumer gets an item from a full buffer, the buffer becomes empty, etc. A producer must wait if there are no empty buffers. Similarly, a consumer must wait if there are no full buffers. Furthermore, waiting processes must be allowed to continue when their awaited events occur.

In the example program, we shall assume that each buffer holds an integer value. The shared global variables are defined as

```
// shared global variables
int buf[NBUF];          // circular buffers
int head, tail;         // indices
int data;               // number of full buffers
```

The buffers are used as a set of circular buffers. The index variables **head** is for putting an item into an empty buffer, and **tail** is for taking an item out of a full buffer. The variable data is the number of full buffers. To support cooperation between producer and consumer, we define a mutex and two condition variables.

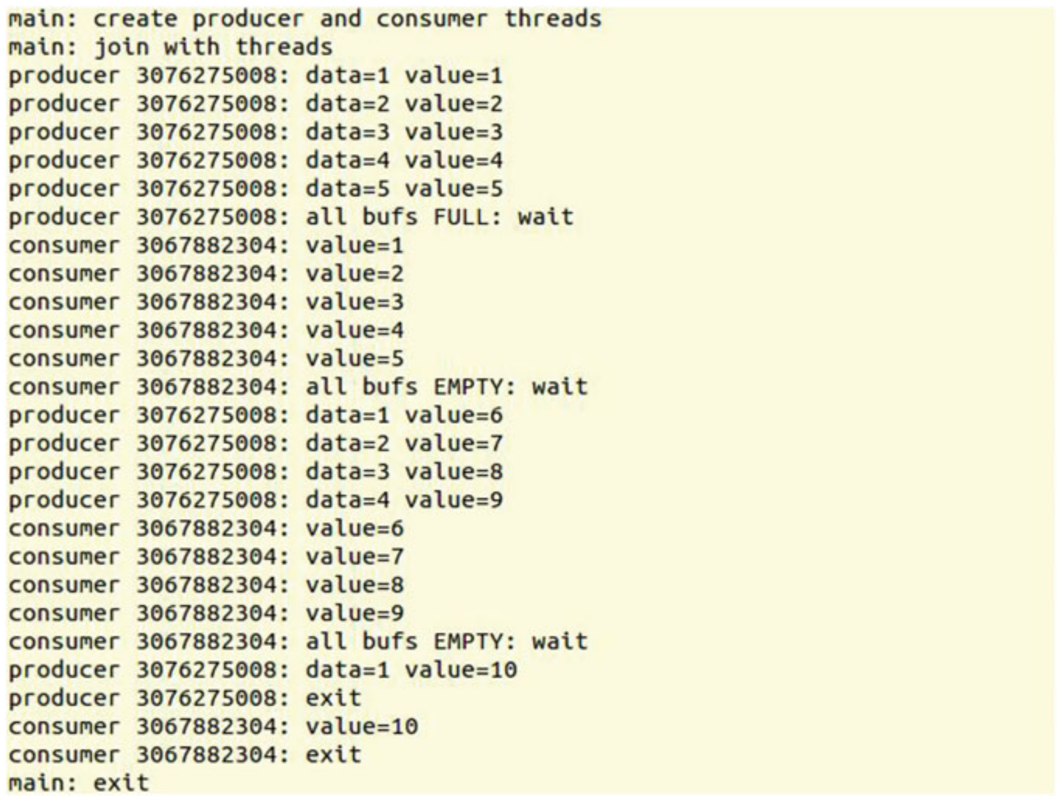
```
     pthread mutex t mutex;       // mutex lock
     pthread cond t empty, full;  // condition variables
```

where **empty** represents the condition of any empty buffers, and **full** represents the condition of any full buffers. When a producer finds there are no empty buffers, it waits on the **empty** condition variable, which is signaled whenever a consumer has consumed a full buffer. Likewise, when a consumer finds there are no full buffers, it waits on the **full** condition variable, which is signaled whenever a producer puts an item into an empty buffer.

The program starts with the main thread, which initializes the buffer control variables and the condition variables. After initialization, it creates a producer and a consumer thread and waits for the threads to join. The buffer size is set to NBUF=5 but the producer tries to put N=10 items into the buffer area, which would cause it to wait when all the buffers are full. Similarly, the consumer tries to get N=10 items from the buffers, which would cause it to wait when all the buffers are empty. In either case, a waiting thread is signaled up by another thread when the awaited conditions are met. Thus, the two threads cooperate with each other through the condition variables. The following shows the program code of Example Program C4.4, which implements a simplified version of the producer-consumer problem with only one producer and one consumer.

```
/* C4.4.c: producer-consumer by threads with condition variables */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NBUF     5
#define N       10

// shared global variables
int buf[NBUF];          // circular buffers
int head, tail;         // indices
int data;               // number of full buffers
pthread mutex t mutex;       // mutex lock
pthread cond t empty, full;  // condition variables

int init()
{
  head = tail = data = 0;
  pthread mutex init(&mutex, NULL);
  pthread cond init(&fullBuf, NULL);
  pthread cond init(&emptyBuf, NULL);
}

void *producer()
{
  int i;
  pthread t me = pthread self();
  for (i=0; i<N; i++){ // try to put N items into buf[ ]
      pthread mutex lock(&mutex);    // lock mutex
      if (data == NBUF){
          printf ("producer %lu: all bufs FULL: wait\n", me);
          pthread cond wait(&empty, &mutex); // wait
      }
      buf[head++] = i+1;              // item = 1,2,..,N
      head %= NBUF;                   // circular bufs
      data++;                         // inc data by 1
      printf("producer %lu: data=%d value=%d\n", me, bp >data, i+1);
      pthread mutex unlock(&mutex);  // unlock mutex
      pthread cond signal(&full);    // unblock a consumer, if any
  }
  printf("producer %lu: exit\n", me);
}

void *consumer()
{
  int i, c;
  pthread t me = pthread self();
  for (i=0; i<N; i++) {
      pthread mutex lock(&mutex);      // lock mutex
      if (data == 0) {
         printf ("consumer %lu: all bufs EMPTY: wait\n", me);
         pthread cond wait(&full, &mutex); // wait
      }
      c = buf[tail++];                 // get an item
      tail %= NBUF;
      data  ;                          // dec data by 1
      printf("consumer %lu: value=%d\n", me, c);
      pthread mutex unlock(&mutex);    // unlock mutex
      pthread cond signal(&empty);     // unblock a producer, if any
  }
  printf("consumer %lu: exit\n", me);
}

int main ()
{
  pthread t pro, con;
  init();
  printf("main: create producer and consumer threads\n");
  pthread create(&pro, NULL, producer, NULL);
  pthread create(&con, NULL, consumer, NULL);
  printf("main: join with threads\n");
  pthread join(pro, NULL);
  pthread join(con, NULL);
  printf("main: exit\n");
}
```

Figure 4.4 shows the output of running the producer-consumer example program.



```
main: create producer and consumer threads
main: join with threads
producer 3076275008: data=1 value=1
producer 3076275008: data=2 value=2
producer 3076275008: data=3 value=3
producer 3076275008: data=4 value=4
producer 3076275008: data=5 value=5
producer 3076275008: all bufs FULL: wait
consumer 3067882304: value=1
consumer 3067882304: value=2
consumer 3067882304: value=3
consumer 3067882304: value=4
consumer 3067882304: value=5
consumer 3067882304: all bufs EMPTY: wait
producer 3076275008: data=1 value=6
producer 3076275008: data=2 value=7
producer 3076275008: data=3 value=8
producer 3076275008: data=4 value=9
consumer 3067882304: value=6
consumer 3067882304: value=7
consumer 3067882304: value=8
consumer 3067882304: value=9
consumer 3067882304: all bufs EMPTY: wait
producer 3076275008: data=1 value=10
producer 3076275008: exit
consumer 3067882304: value=10
consumer 3067882304: exit
main: exit
```

Fig. 4.4 Outputs of Producer Consumer Program

4.6.5 Semaphores

Semaphores are general mechanisms for process synchronization. A (counting) semaphore is a data structure

```
struct sem{
   int value;              // semaphore (counter) value;
   struct process *queue   // a queue of blocked processes
}s;
```

Before using, a semaphore must be initialized with an initial value and an empty waiting queue. Regardless of the hardware platform, i.e. whether on single CPU systems or multiprocessing systems, the low level implementation of semaphores guarantees that each semaphore can only be operated by one executing entity at a time and operations on semaphores are **atomic** (indivisible) or **primitive** from the viewpoint of executing entities. The reader may ignore such details here and focus on the high level operations on semaphores and their usage as a mechanism for process synchronization. The most well-known operations on semaphores are P and V (Dijkstra 1965), which are defined as follows.

```
P(struct sempahore *s)          |   V(struct semaphore *s)
{                               |   {
  s >value  ;                   |      s >value++;
  if (s >value < 0)             |      if (s >value <= 0)
     BLOCK(s);                  |         SIGNAL(s);
}                               |   }
```

where BLOCK(s) blocks the calling process in the semaphore's waiting queue, and SIGNAL (s) unblocks a process from the semaphore's waiting queue.

Semaphores are not part of the original Pthreads standard. However, most Pthreads now support semaphores of POSIX 1003.1b. POSIX semaphores include the following functions

```
int sem init(sem, value) : initialize sem with an initial value
int sem wait(sem)        : similar to P(sem)
int sem post(sem)        : similar to V(sem)
```

The major difference between semaphores and condition variables is that the former includes a counter, manipulate the counter, test the counter value to make decisions, etc. all in the Critical Region of atomic or primitive operations, whereas the latter requires a specific mutex lock to enforce the Critical Region. In Pthreads, mutexes are strictly for locking and condition variables are for threads cooperation. In contrast, counting semaphores with initial value 1 can be used as locks. Semaphores with other initial values can be used for cooperation. Therefore, semaphores are more general and flexible than condition variables. The following example illustrates the advantages of semaphores over condition variables.

Example 4.5: The producer-consumer problem can be solved more efficiently by using semaphores. In this example, **empty=N and full=0** are semaphores for producers and consumers to cooperate with one another, and **mutex =1** is a lock semaphore for processes to access shared buffers one at a time in a Critical Region. The following shows the pseudo code of the producer-consumer problem using semaphores.

```
   ITEM buf[N];           // N buffers of ITEM type
   int head=0, tail=0;    // buffer indices
   struct semaphore empty=N, full=0, mutex=1; // semaphores

--------- Producer ------------- Consumer------------
   while(1){                |   while(1){
     produce an item;       |      ITEM item;
     P(&empty);             |      P(&full);
     P(&mutex);             |      P(&mutex)
       buf[head++]=item;    |        item=buf[tail++];
       head %= N;           |        tail %= N;
     V(&mutex);             |      V(&mutex);
     V(&full);              |      V(&empty);
    }                       |   }
-----------------------------------------------------
```

4.6.6 Barriers

The threads join operation allows a thread (usually the main thread) to wait for the termination of other threads. After all awaited threads have terminated, the main thread may create new threads to continue executing the next parts of a parallel program. This requires the overhead of creating new threads. There are situations in which it would be better to keep the threads alive but require them not to go on until all of them have reached a prescribed point of synchronization. In Pthreads, the mechanism is the **barrier**, along with a set of barrier functions. First, the main thread creates a barrier object

```
pthread barrier t barrier;
```

and calls

```
pthread barrier init(&barrier NULL, nthreads);
```

to initialize it with the number of threads that are to be synchronized at the barrier. Then the main thread creates working threads to perform tasks. The working threads use

```
pthread barrier wait( &barrier)
```

to wait at the barrier until the specified number of threads have reached the barrier. When the last thread arrives at the barrier, all the threads resume execution again. In this case, a barrier acts as a rendezvous point, rather than as a graveyard, of threads. We illustrate the use of barriers by an example.

4.6.7 Solve System of Linear Equations by Concurrent Threads

We demonstrate applications of concurrent threads and threads join and barrier operations by an example.

Example 4.6: The example is to solve a system of linear equations by concurrent threads. Assume AX = B is a system of linear equations, where A is an N × N matrix of real numbers, X is a column vector of N unknowns and B is column vector of constants. The problem is to compute the solution vector X. The most well known algorithm for solving systems of linear equations is **Gauss elimination**. The algorithm consists of 2 major steps; **row reduction**, which reduces the combined matrix [A|B] to an upper-triangular form, followed by **back substitution**, which computes the solution vector X. In the row-reduction steps, **partial pivoting** is a scheme which ensures the leading element of the row used to reduce other rows has the maximum absolute value. Partial pivoting helps improve the accuracy of the numerical computations. The following shows a Gauss elimination algorithm with partial pivoting.

/****** Gauss Elimination Algorithm with Partial Pivoting *******/**
Step 1: Row reduction: reduce [A|B] to upper triangular form

```
      for (i=0; i<N ; i++){        // for rows i = 0 to N 1
        do partial pivoting;       // exchange rows if needed
 (1).   // barrier
        for (j=i+1; j<=N; j++){    // for rows j = i+1 to N
          for (k=i+1; k<=N; k++){ // for columns k = i+1 to N
              f = A[j,i]/A[i,i];  // reduction factor
              A[j,k]  = A[j,k]*f; // reduce row j
          }
          A[j][i] = 0;            // A[j,i] = 0
        }
 (2).   // barrier
      }
 (3). // join
```

Step 2: Back Substitution: compute xN-1, xN-2, . . . , x0 in that order

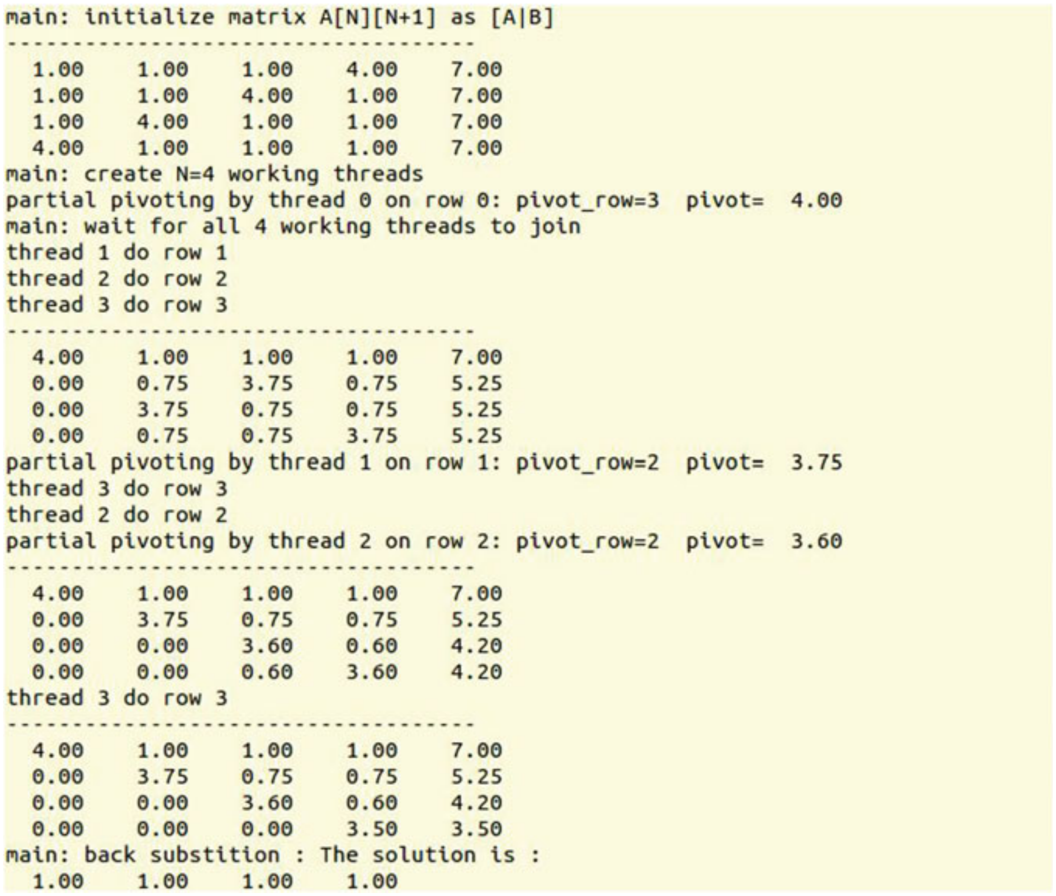
The Gauss elimination algorithm can be parallelized as follows. The algorithm starts with a main thread, which creates N working threads to execute the **ge(thread id)** function and waits for all the working threads to join. The threads function ge() implements the row reduction step of the Gauss elimination algorithm. In the ge() function, for each iteration of row = 0 to N-2, the thread with thread ID=i does the partial pivoting on a corresponding row i. All other threads wait at a barrier (1) until partial pivoting is complete. Then each working thread does row reduction on a unique row equal to its ID number. Since all working threads must have finished the current row reductions before iteration on the next row can start, so they wait at another barrier (2). After reducing the matrix [A|B] to upper triangular form, the final step is to compute the solutions x[N-i], for i=1 to N in that order, which is inherently sequential. The main thread must wait until all the working threads have ended before starting the back substitution. This is achieved by join operations by the main thread. The following shows the complete code of the Example Program C4.5.

```
/** C4.5.c: Gauss Elimination with Partial Pivoting **/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>

#define N 4
double A[N][N+1];
pthread barrier t barrier;

int print matrix()
{
   int i, j;
   printf("                                  \n");
   for(i=0; i<N; i++){
       for(j=0;j<N+1;j++)
          printf("%6.2f  ",  A[i][j]);
       printf("\n");
   }
}

void *ge(void *arg) // threads function: Gauss elimination
{
  int i, j, prow;
  int myid = (int)arg;
  double temp, factor;
  for(i=0; i<N 1; i++){
     if (i == myid){
        printf("partial pivoting by thread %d on row %d: ", myid, i);
        temp = 0.0; prow = i;
        for (j=i; j<=N; j++){
            if (fabs(A[j][i]) > temp){
               temp = fabs(A[j][i]);
               prow = j;
            }
```

```
      }
      printf("pivot row=%d  pivot=%6.2f\n", prow, A[prow][i]);
      if (prow != i){  // swap rows
          for (j=i; j<N+1; j++){
              temp = A[i][j];
              A[i][j] = A[prow][j];
              A[prow][j] = temp;
          }
      }
   }
   // wait for partial pivoting done
   pthread barrier wait(&barrier);
   for(j=i+1; j<N; j++){
       if (j == myid){
           printf("thread %d do row %d\n", myid, j);
           factor = A[j][i]/A[i][i];
           for (k=i+1; k<=N; k++)
                A[j][k]  = A[i][k]*factor;
           A[j][i] = 0.0;
       }
   }
   // wait for current row reductions to finish
   pthread barrier wait(&barrier);
   if (i == myid)
       print matrix();
   }
}

int main(int argc, char *argv[])
{
   int i, j;
   double sum;
   pthread t threads[N];

   printf("main: initialize matrix A[N][N+1] as [A|B]\n");
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
          A[i][j] = 1.0;
   for (i=0; i<N; i++)
        A[i][N i 1] = 1.0*N;
   for (i=0; i<N; i++){
        A[i][N] = 2.0*N   1;
   }
   print matrix();  // show initial matrix [A|B]

   pthread barrier init(&barrier, NULL, N); // set up barrier

   printf("main: create N=%d working threads\n", N);
   for (i=0; i<N; i++){
       pthread create(&threads[i], NULL, ge, (void *)i);
   }
```

```
     printf("main: wait for all %d working threads to join\n", N);
     for (i=0; i<N; i++){
          pthread join(threads[i], NULL);
     }
     printf("main: back substitution : ");
     for (i=N 1; i>=0; i  ){
         sum = 0.0;
         for (j=i+1; j<N; j++)
             sum += A[i][j]*A[j][N];
             A[i][N] = (A[i][N]  sum)/A[i][i];
     }
     // print solution
     printf("The solution is :\n");
     for(i=0; i<N; i++){
         printf("%6.2f  ", A[i][N]);
     }
     printf("\n");
}
```

Figure 4.5 shows the sample outputs of running the Example 4.6 program, which solves a system of equations with N=4 unknowns by Gauss elimination with partial pivoting.



```
main: initialize matrix A[N][N+1] as [A|B]
-----------------------------------
  1.00    1.00    1.00    4.00    7.00
  1.00    1.00    4.00    1.00    7.00
  1.00    4.00    1.00    1.00    7.00
  4.00    1.00    1.00    1.00    7.00
main: create N=4 working threads
partial pivoting by thread 0 on row 0: pivot_row=3  pivot=  4.00
main: wait for all 4 working threads to join
thread 1 do row 1
thread 2 do row 2
thread 3 do row 3
-----------------------------------
  4.00    1.00    1.00    1.00    7.00
  0.00    0.75    3.75    0.75    5.25
  0.00    3.75    0.75    0.75    5.25
  0.00    0.75    0.75    3.75    5.25
partial pivoting by thread 1 on row 1: pivot_row=2  pivot=  3.75
thread 3 do row 3
thread 2 do row 2
partial pivoting by thread 2 on row 2: pivot_row=2  pivot=  3.60
-----------------------------------
  4.00    1.00    1.00    1.00    7.00
  0.00    3.75    0.75    0.75    5.25
  0.00    0.00    3.60    0.60    4.20
  0.00    0.00    0.60    3.60    4.20
thread 3 do row 3
-----------------------------------
  4.00    1.00    1.00    1.00    7.00
  0.00    3.75    0.75    0.75    5.25
  0.00    0.00    3.60    0.60    4.20
  0.00    0.00    0.00    3.50    3.50
main: back substitution : The solution is :
  1.00    1.00    1.00    1.00
```

Fig. 4.5 Sample Outputs of Example 4.6 Program

4.6.8 Threads in Linux

Unlike many other operating systems, Linux does not distinguish processes from threads. To the Linux kernel, a thread is merely a process that shares certain resources with other processes. In Linux both processes and threads are created by the clone() system call, which has the prototype

```
int clone(int (*fn)(void *), void *child stack, int flags, void *arg)
```

As can be seen, clone() is more like a thread creation function. It creates a child process to execute a function fn(arg) with a child stack. The flags field specifies the resources to be shared by the parent and child, which includes

CLONE VM: parent and child share address space
CLONE FS: parent and child share file system information, e.g. root. CWD
CLONE FILES: parent and child share opened files
CLONE SIGHAND: parent and child share signal handlers and blocked signals

If any of the flags is specified, both processes share exactly the SAME resource, not a separate copy of the resource. If a flag is not specified, the child process usually gets a separate copy of the resource. In this case, changes made to a resource by one process do not affect the resource of the other process. The Linux kernel retains fork() as a system call but it may be implemented as a library wrapper that calls clone() with appropriate flags. An average user does not have to worry about such details. It suffices to say that Linux has an efficient way of supporting threads. Moreover, most current Linux kernels support Symmetric Multiprocessing (SMP). In such Linux systems, processes (threads) are scheduled to run on multiprocessors in parallel.

4.7 Programming Project: User-Level Threads

The programming project is to implement **user-level threads** to simulate threads operations in Linux. The project consists of 4 parts. Part 1 presents the base code of the project to help the reader get started. The base code implements a multitasking system, which supports independent executions of tasks within a Linux process. It is the same MT multitasking system presented in Chap. 3, but adapted to the user-level threads environment. Part 2 is to extend the base code to implement support for task join operation. Part 3 is to extend the base code to support mutex operations. Part 4 is to implement counting semaphores to support task cooperation, and demonstrate semaphores in the multitasking system.

4.7.1 Project Base Code: A Multitasking System

Programming Project PART 1: Base Code: The following shows the base code of the programming project, which will be explained in latter sections.