Example: Using Mutexes

- This example program illustrates the use of mutex variables in a Pthreads program that performs a dot product.
- The main data is made available to all threads through a globally accessible structure.
- Each thread works on a different part of the data.
- The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

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

/* The following structure contains the necessary information
 * to allow the function "dotprod" to access its input data and
 * place its output into the structure.
 */

typedef struct
{
   double *a;
   double *b;
   double sum;
   int    veclen;
} DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100

DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

/* The function dotprod is activated when the thread is created.
 * All input to this routine is obtained from a structure
 * of type DOTDATA and all output from this function is written into
 * this structure. The benefit of this approach is apparent for the
 * multi-threaded program: when a thread is created we pass a single
 * argument to the activated function - typically this argument
 * is a thread number. All  the other information required by the
 * function is accessed from the globally accessible structure.
 */

void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */
```

```
    int i, start, end, len;
    long offset;
    double mysum, *x, *y;
    offset = (long)arg;

    len = dotstr.veclen;
    start = offset * len;
    end   = start + len;
    x = dotstr.a;
    y = dotstr.b;

    /*
     Perform the dot product and assign result
     to the appropriate variable in the structure.
     */

    mysum = 0;
    for (i = start; i < end ; i++) {
       mysum += (x[i] * y[i]);
    }

    /*
     Lock a mutex prior to updating the value in the shared
     structure, and unlock it upon updating.
     */
    pthread_mutex_lock(&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock(&mutexsum);

    pthread_exit((void*) 0);
 }

 /* The main program creates threads which do all the work and then
  * print out result upon completion. Before creating the threads,
  * the input data is created. Since all threads update a shared structure,
  * we need a mutex for mutual exclusion. The main thread needs to wait for
  * all threads to complete, it waits for each one of the threads. We specify
  * a thread attribute value that allow the main thread to join with the
  * threads it creates. Note also that we free up handles when they are
  * no longer needed.
  */

 int main (int argc, char *argv[])
 {
    long i;
    double *a, *b;
    void *status;
    pthread_attr_t attr;

    /* Assign storage and initialize values */
    a = (double*) malloc (NUMTHRDS * VECLEN * sizeof(double));
    b = (double*) malloc (NUMTHRDS * VECLEN * sizeof(double));

    for (i = 0; i < VECLEN * NUMTHRDS; i++) {
       a[i] = 1.0;
       b[i] = a[i];
    }

    dotstr.veclen = VECLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum = 0;

    pthread_mutex_init(&mutexsum, NULL);

    /* Create threads to perform the dot product  */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

```
   for(i = 0; i < NUMTHRDS; i++) {
      /* Each thread works on a different set of data. The offset is specified
      * by 'i'. The size of the data for each thread is indicated by VECLEN.
      */
      pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

   pthread_attr_destroy(&attr);

   /* Wait on the other threads */
   for(i = 0; i < NUMTHRDS; i++) {
      pthread_join(callThd[i], &status);
   }

   /* After joining, print out the results and cleanup */
   printf("Sum =  %f\n", dotstr.sum);
   free(a);
   free(b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

Serial version: source

Parallel version: source