```
/* Sample program demonstrating POSIX threads */
/* Topics covered:
   (1) Thread creation, parameter passing, attribute setting, thread joining
   (2) Using mutexes for critical sections
   (3) Using condition variables for synchronization
*/

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

#include <string.h>   /* Needed for strcpy() */
#include <time.h>     /* Needed for time() to seed the RNG */
#include <unistd.h>   /* Needed for sleep() and usleep() */
#include <pthread.h>  /* Needed for all pthread library calls */

/* Number of worker threads */
#define N 4

/* Data type for parameters to be passed to worker threads during start up */
typedef struct {
   int tno;
   char tname[5];
} tinfo;

/* Array size */
#define S 100

/* Global arrays to be shared by all threads */
int A[S], C[S];

/* mutex for mutually exclusive updating of the arrays A[] and C[] */
pthread_mutex_t csmutex;

/* mutex and condition variables for winding up */
pthread_mutex_t donemutex;
pthread_cond_t donecond;
int mdone = 0, wdone = 0;

/* This is the main function for a worker thread */
/* A worker thread receives a number and a 4-letter name via targ */
void *tmain ( void *targ )
{
   /* Local variables are not shared with other threads */
   int no, i;
   char name[5];
   pthread_t tid;
   int count = 0, s, t;

   /* Retrieve my number and name from the parameter passed */
   no = ((tinfo *)targ) -> tno;
   strcpy(name,((tinfo *)targ) -> tname);
```

```
/* Retrieve my thread id */
tid = pthread_self();

printf("\t\t\t\t\t(%d,%s) [%lu] running\n", no, name, tid);

while (1) {
  /* Check for termination condition */

  pthread_mutex_lock(&donemutex);

  /* if the master thread is done */
  if (mdone) {
    ++wdone;

    if (wdone < N) {
      /* Wait for signal from the last worker thread to finish */
      /* This atomically unlocks the donemutex too */

      if (wdone == 1) printf("\n");
      printf("\t\t\t\t\t(%d,%s) going to wait\n", no, name);
      pthread_cond_wait(&donecond, &donemutex);

      /* When the signal is received, donemutex is again locked,
        and must be explicitly freed. */
    } else {
      /* This is the last worker thread to finish. It must not itself
        wait on the condition variable. Instead, it should wake up
        the other worker threads waiting on the condition variable. */

      printf("\n");
      printf("\t\t\t\t\t(%d,%s) going to broadcast\n", no, name);
      printf("\n");
      pthread_cond_broadcast(&donecond);

      /* Another option is to call pthread_cond_signal(&donecond)
        N - 1 times in order to signal the remaining threads one
        by one. */
    }

    pthread_mutex_unlock(&donemutex);

    /* Explicitly exit */
    printf("\t\t\t\t\t(%d,%s) exits with count = %d\n", no, name, count);
    pthread_exit(NULL);
  }

  /* The master thread is still sleeping, so I continue to work */
  pthread_mutex_unlock(&donemutex);

  i = rand() % S;

  /* Entering critical section */
```

```
    pthread_mutex_lock(&csmutex);       /* Lock mutex for critical section */
    s = A[i];                             /* Read A[i] */
    t = (s % 2 == 0) ? (s / 2) : (3 * s + 1);        /* Compute new value */
    A[i] = t;                           /* Update A[i] */
    ++C[i];                              /* Update C[i] */
    pthread_mutex_unlock(&csmutex);    /* Unlock mutex for critical section */

    /* Leaving critical section */

    /* Update and print local values only */
    ++count;
    printf("\t\t\t\t\t(%d,%s) changes A[%2d] : %5d -> %5d\n", no,name,i,s,t);

    usleep(10);
  }
}

/* Initialize the arrays A[] and C[]. This is done before worker threads are
  created, so there is no necessity to use a mutex in this function. */
void init_arrays ( )
{
  int i;

  for (i=0; i<S; ++i) {
    A[i] = 1 + rand() % 999;     /* A random integer between 1 and 999 */
    C[i] = 0;                  /* Count of changes done on the ith entry */
  }
}

/* Print the arrays A and C. This also does not require mutual exclusion. */
void print_arrays ( )
{
  int i;

  for (i=0; i<S; ++i) {
    printf("%4d (%4d)", A[i], C[i]);
    if (i % 7 == 6) printf("\n");
  }
  printf("\n");
}

/* Function to create N worker threads. Their thread ids are stored in tid.
  The parameters (numbers and names) are stored in param. */
void create_workers ( pthread_t *tid, tinfo *param )
{
  pthread_attr_t attr;
  int i, j;

  /* Set attributes for creating joinable threads */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  /* Create worker threads in a loop */
  for (i=0; i<N; ++i) {
    /* Set the number of the thread */
    param[i].tno = i + 1;

    /* Set a random 4-letter name for the thread */
    for (j=0; j<4; ++j) param[i].tname[j] = 'A' + rand() % 26;
    param[i].tname[4] = '\0';

    /* Create thread with number and name passed as parameters */
    if  (pthread_create(tid + i, &attr, tmain, (void *)(param+i))) {
      fprintf(stderr, "Master thread: Unable to create thread\n");
      pthread_attr_destroy(&attr);
      exit(1);
    }

    printf("(%d,%s) [%lu] created\n", param[i].tno, param[i].tname, tid[i]);
  }

  /* Wait for a while to insure that all worker threads get the chance to be
    scheduled and read the correct local values defined in this function */
  sleep(1);

  pthread_attr_destroy(&attr);
}

void create_mutex ()
{
  /* Initialize mutex variables. This can also be done statically using
    mutex = PTHREAD_MUTEX_INITIALIZER; */
  pthread_mutex_init(&csmutex, NULL);
  pthread_mutex_init(&donemutex, NULL);

  /* At this point, our mutexes should be unlocked, but this behavior
    may be system-dependent. For portability, we add the following
    lines that force the mutexes to unlocked states, irrespective of
    whether they were locked earlier or not. */

  pthread_mutex_trylock(&csmutex);   /* Try to lock mutex (non-blocking) */
  pthread_mutex_unlock(&csmutex);    /* Now, unlock the mutex */

  pthread_mutex_trylock(&donemutex); /* Try to lock mutex (non-blocking) */
  pthread_mutex_unlock(&donemutex);  /* Now, unlock the mutex */

  /* Initialize condition variable */
  pthread_cond_init(&donecond, NULL);
}

void do_work ( pthread_t *tid, tinfo *param )
{
  int i;
```

```
   /* Simulate some work for ten seconds */
   sleep(10);

   /* At the end of work, the master thread sets the mdone flag */
   pthread_mutex_lock(&donemutex);
   mdone = 1;
   pthread_mutex_unlock(&donemutex);

   /* Meanwhile the worker threads finish one by one */
   /* The master waits for all the worker threads to finish */
   for (i=0; i<N; ++i) {
      if (pthread_join(tid[i],NULL)) {
         fprintf(stderr, "Unable to wait for thread [%lu]\n", tid[i]);
      } else {
         printf("(%d,%s) has joined\n", param[i].tno, param[i].tname);
      }
   }
   printf("\n");
}

void wind_up ()
{
   /* Destroy mutex and condition variables */
   printf("\nWinding up\n\n");
   pthread_mutex_destroy(&csmutex);
   pthread_mutex_destroy(&donemutex);
   pthread_cond_destroy(&donecond);
}

int main ()
{
   pthread_t tid[N];
   tinfo param[N];

   srand((unsigned int)time(NULL));
   create_mutex();
   init_arrays();
   print_arrays();
   create_workers(tid,param);
   do_work(tid,param);
   print_arrays();
   wind_up();
   exit(0);
}

/* End of program */
```