Practical Course on Parallel Computing
Hands-on Exercise

Dr. Gen Kawamura

April 16, 2018

1 Posix Thread: Basis

Let's start Pthreads.

The following codes can generate the first Pthreads examples in the lecture. Write and compile the codes and run the executables. [1]

- Hello World *basic.c*

```
#include <stdio.h>
#include <pthread.h>
void *hello()
{
  printf("Hello␣World.\n");
  pthread_exit(NULL);
}
main () {
  pthread_t thread;
  pthread_create(&thread, NULL, hello, NULL);
  pthread_exit(0);
}
```

- Excecusion of basic.c

```
$ gcc -pthread basic.c -o basic
$ ./basic
```

- With pthread arguments: *answer.c*

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

[1] The codes will be available on the lecture page later - :https://projects.gwdg.de/projects/practical-course-on-parallel-computing/wiki/wiki

```
#include<pthread.h>

void *answer(void *value)
{
  long number = (long ) value;
  printf("The␣answer␣is␣%ld.\n", number);
  pthread_exit(NULL);
}
main () {
long value = 42;
pthread_t thread;
pthread_create(&thread, NULL, answer, (void *) value);
pthread_exit(0);
}
```

- With many pthread arguments: *hello_arg2.c*

```
/******************************************************
* FILE: hello_arg2.c
* DESCRIPTION:
*   A "hello world" Pthreads program which demonstrates
    another safe way
*   to pass arguments to threads during thread creation.  In
     this case,
*   a structure is used to pass multiple arguments.
* AUTHOR: Blaise Barney
* LAST REVISED: 01/29/09
******************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

struct thread_data
{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;

   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread␣%d:␣%s␣␣Sum=%d\n", taskid, hello_msg, sum)
       ;
   pthread_exit(NULL);
```

```
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int *taskids[NUM_THREADS];
  int rc, t, sum;

  sum=0;
  messages[0] = "English:␣Hello␣World!";
  messages[1] = "French:␣Bonjour,␣le␣monde!";
  messages[2] = "Spanish:␣Hola␣al␣mundo";
  messages[3] = "Klingon:␣Nuq␣neH!";
  messages[4] = "German:␣Guten␣Tag,␣Welt!";
  messages[5] = "Russian:␣Zdravstvytye,␣mir!";
  messages[6] = "Japan:␣Sekai␣e␣konnichiwa!";
  messages[7] = "Latin:␣Orbis,␣te␣saluto!";

  for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id  = t;
    thread_data_array[t].sum        = sum;
    thread_data_array[t].message    = messages[t];
    printf("Creating␣thread␣%d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
                        (void *)  &thread_data_array[t]);
    if (rc) {
      printf("ERROR;␣return␣code␣from␣pthread_create()␣is␣%d
          \n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

2 Shared Data

The global C variables in the following codes are global over thread boundaries.

- Basic example for shared data: *basic_glob_var.c*

```
#include <stdio.h>
#include <pthread.h>
int answer=42;
void *hello()
{
  printf("The␣answer␣is␣again␣%d\n",answer);
  pthread_exit(NULL);
}
int main () {
  pthread_t thread;
  pthread_create(&thread, NULL, hello, NULL);
  pthread_create(&thread, NULL, hello, NULL);
  pthread_exit(0);
}
```

- A mutex example using dotprod (serial ver.): *dotprod_serial.c*

```
/*****************************************************
* FILE: dotprod_serial.c
* DESCRIPTION:
*   This is a simple serial program which computes the dot
    product of two
*   vectors.  The threaded version can is dotprod_mutex.c.
* SOURCE: Vijay Sonnad, IBM
* LAST REVISED: 01/29/09 Blaise Barney
*****************************************************/
#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 so that it can be accessed later.
*/

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

#define VECLEN 100000

DOTDATA dotstr;

/*
We will use a function (dotprod) to perform the scalar
    product.
All input to this routine is obtained through a structure of
type DOTDATA and all output from this function is written
    into
this same structure.  While this is unnecessarily
    restrictive
for a sequential program, it will turn out to be useful when
we modify the program to compute in parallel.
*/

void dotprod()
{

/* Define and use local variables for convenience */

   int start, end, i;
   double mysum, *x, *y;

   start=0;
   end = dotstr.veclen;
   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]);
  }
  dotstr.sum = mysum;

}

/*
The main program initializes data and calls the dotprd()
    function.
Finally, it prints the result.
*/

int main (int argc, char *argv[])
{
  int i,len;
  double *a, *b;

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

  for (i=0; i<len; i++) {
    a[i]=1;
    b[i]=a[i];
  }

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

  /* Perform the  dotproduct */
  dotprod ();

  /* Print result and release storage */
  printf ("Sum =  %f \n", dotstr.sum);
  free (a);
  free (b);
}
```

- A Mutex example using dotprod (threadable ver.): *dotprod_mutex.c*

```
/*****************************************************
* FILE: dotprod_mutex.c
* DESCRIPTION:
*   This example program illustrates the use of mutex
    variables
*   in a threads program. This version was obtained by
    modifying the
*   serial version of the program (dotprod_serial.c) which
    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
     .
* SOURCE: Vijay Sonnad, IBM
* LAST REVISED: 01/29/09 Blaise Barney
*************************************************/
#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.  This structure is
unchanged from the sequential version.
*/

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

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100000

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

/*
The function dotprod is activated when the thread is created
     .
As before, 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;
   printf("Thread␣%ld␣did␣%d␣to␣%d:␣␣mysum=%f␣global␣sum=%f\
       n",offset,start,end,mysum,dotstr.sum);
   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;
    b[i]=a[i];
  }

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

  pthread_mutex_init(&mutexsum, NULL);


  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], NULL, dotprod, (void *)i);
  }

  /* 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);
}
```

3 Signaling and Condition Variables

With a condition variable it can signal another thread about an event, e.g. that they are finished doing something.

- Signaling and Condition Variables: *condvar.c*

```
/*****************************************************
* FILE: condvar.c
* DESCRIPTION:
*   Example code for using Pthreads condition variables.
    The main thread
*   creates three threads.  Two of those threads increment a
     "count" variable,
*   while the third thread watches the value of "count".
    When "count"
*   reaches a predefined limit, the waiting thread is
    signaled by one of the
*   incrementing threads. The waiting thread "awakens" and
    then modifies
```

```
*    count. The program continues until the incrementing
     threads reach
*    TCOUNT. The main program prints the final value of count
     .
* SOURCE: Adapted from example code in "Pthreads Programming
     ", B. Nichols
*    et al. O'Reilly and Associates.
* LAST REVISED: 10/14/10  Blaise Barney
******************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when
        condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold
          reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking
        mutex\n",
     my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock
        */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);
```

```
  /*
  Lock mutex and wait for signal.  Note that the
      pthread_cond_wait routine
  will automatically and atomically unlock mutex while it
      waits.
  Also, note that if COUNT_LIMIT is reached before this
      routine is run by
  the waiting thread, the loop will be skipped to prevent
      pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count():␣thread␣%ld␣Count=␣%d.␣Going␣into␣
        wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count():␣thread␣%ld␣Condition␣signal␣
        received.␣Count=␣%d\n", my_id,count);
    printf("watch_count():␣thread␣%ld␣Updating␣the␣value␣of␣
        count...\n", my_id,count);
    count += 125;
    printf("watch_count():␣thread␣%ld␣count␣now␣=␣%d.\n",
        my_id, count);
  }
  printf("watch_count():␣thread␣%ld␣Unlocking␣mutex.\n",
      my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  pthread_create(&threads[0], NULL, watch_count, (void *)t1)
      ;
  pthread_create(&threads[1], NULL, inc_count, (void *)t2);
  pthread_create(&threads[2], NULL, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main():␣Waited␣and␣joined␣with␣%d␣threads.␣Final␣
      value␣of␣count␣=␣%d.␣Done.\n",
          NUM_THREADS, count);

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
```

4 Performance - Bug

This example code contains a bug. Please find and fix it.

- A buggy code: Where is the bug? *bug6.c*

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

#define NUMTHRDS 4
#define NUMBERS 12800000000
long sum=0;

struct interval {
  long start;
  long end;
  int tid;
};

typedef struct interval intv;

void *add(void *arg)
{
   intv *myinterval = (intv*) arg;
   long start = myinterval->start;
   long end   = myinterval->end;

   printf("I am thread %d and my interval is %ld to %ld\n",
             myinterval->tid,start,end);
   long j;
   for(j=start;j<=end;j++) {
     sum = sum + j;
   }

   printf("Thread %d finished\n",myinterval->tid);

   pthread_exit((void*) 0);
}



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

intv *intervals;
intervals = (intv*) malloc(NUMTHRDS*sizeof(intv));

void *status;
pthread_t threads[NUMTHRDS];
int i;

for(i=0; i<NUMTHRDS; i++) {
  intervals[i].start = i*(NUMBERS/NUMTHRDS);
  intervals[i].end   = (i+1)*(NUMBERS/NUMTHRDS) - 1;
  intervals[i].tid=i;
  pthread_create(&threads[i], NULL, add, (void *) &intervals
      [i]);
}
```

```
/* Wait on the threads for final result */
for(i=0; i<NUMTHRDS; i++)
  pthread_join(threads[i], &status);

/* After joining, print out the results and cleanup */
printf ("Final␣Global␣Sum=%li\n",sum);
free (intervals);
pthread_exit(NULL);
}
```