POSIX THREADS PROGRAMMING SUMMARY WITH EXERCISE NOTES

An hello.c Example

1. Create, compile and run a Pthreads "Hello world" program
2. Main program creates several threads, each of which executes a "print hello" thread routine. The argument passed to that routine is their thread ID.
3. The thread's "print hello" routine accepts the thread ID argument and prints "hello world from thread #". Then it calls pthread_exit to finish. Main program calls pthread_exit as the last thing it does.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS    5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }
```

```
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Notes:

- Pthread_create only receive pointer type arguments.
- By calling pthread_exit(NULL) before main() finished
- it will block and be kept alive to support the threads
- unti they are done.

Argument Passing

1. Review the hello_arg2.c example codes. Notice how the how to pass multiple arguments through a structure.
2. Compile and run both programs, and observe output.

```
#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);
}
```

3. Now review, compile and run the bug3.c program. What's wrong? How would you fix it?

   below is the part of the code with bug:

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

bug:

- t□□□□□□□□□□□□□□□□□□□□□□□thread□□□□□□□
- □□thread_n□□□□□□□□□□main□□for loop□□□
- t is a temporary variable for each thread ID,
- by passinf its address, every thread can modify it,

- Thus it is possible that before print ID in each thread,
- t is already changed by main loop.

Note:

- using thread-type and argument-structure-type array
- is good for tracking and accessing data

Thread Exiting

1. Review, compile (for gcc include the -lm flag) and run the bug5.c program.
2. What happens? Why? How would you fix it?

```
int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0;t<NUM_THREADS;t++){
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }
    //□□□ pthread_exit□NULL□; main()□□□□□□□□□□□,□□□□□□□□□
    printf("Main: Done.\n");
}
```

bug:

- without pthread_exit(),
- main() terminate the whole process even threads are still working.

Stack Management

1. Review, compile and run the bug2.c program.
2. What happens? Why? How would you fix it?

```
int main(int argc, char *argv[])
{
  pthread_t threads[NTHREADS];
  size_t stacksize;
  pthread_attr_t attr; ///attr□□□□□□□□□□□□hello□□□□
  int rc;
  long t;
  pthread_attr_init(&attr);
  pthread_attr_getstacksize (&attr, &stacksize);
  printf("Thread stack size = %li bytes (hint, hint)\n",stacksize);
  //stack□□
  //stack_size = sizeof(double)*ARRAY_SIZE + EXTRA ;
  //pthread_attr_setstacksize(&attr,stack_size);
```

```
  for (t = 0; t < NTHREADS; t++)
  {
    rc = pthread_create(&threads[t], NULL, Hello, (void *)t); //attr □□pass to
thread_create
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
  }
  printf("Created %ld threads.\n", t);
  pthread_exit(NULL);
}
```

bug:

- stack□□,
- stack_size = sizeof(double)*ARRAY_SIZE + EXTRA;
- pthread_attr_setstacksize(&attr,stack_size);

Things to note:

- attr variable and its scoping
- use of the pthread_attr_setstacksize routine
- initialization of the attr variable with pthread_attr_init
- passing the attr variable to pthread_create

Mutexes

1. Review, compile and run the dotprod_serial.c program. As its name implies, it is serial - no threads are created.
2. Now review, compile and run the dotprod_mutex.c program. This version of the dotprod program uses threads and requires a mutex to protect the global sum as each thread updates it with their partial sums.

```
#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);

/* Create threads to perform the dotproduct  */
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);
}
```

3. Execute the dotprod_mutex program several times and notice that the order in which threads update the global sum varies.
4. Review, compile and run the bug6.c program.

```
void *dotprod(void *arg)
{
  /* Each thread works on a different set of data.
   * The offset is specified by the arg parameter. The size of
   * the data for each thread is indicated by VECLEN.
   */
   int i, start, end, offset, len;
   long tid = (long)arg;
   offset = tid;
   len = VECLEN;
   start = offset*len;
   end   = start + len;

/* Perform my section of the dot product */
   printf("thread: %ld starting. start=%d end=%d\n",tid,start,end-1);
   for (i=start; i<end ; i++)
      sum += (a[i] * b[i]);
   printf("thread: %ld done. Global sum now is=%li\n",tid,sum);

   pthread_exit((void*) 0);
}
```

bug:

- in this thread function snippet
- it update sum variable without mutex

Condition Variables

1. Review, compile and run the condvar.c program. This example is essentially the same as the shown in the tutorial. Observe the output of the three threads.
2. Now, review, compile and run the bug1.c program. Observe the output of the five threads. What happens? See if you can determine why and fix the problem. The explanation is provided in the bug examples table above, and an example solution is provided by the bug1fix.c program.

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

#define NUM_THREADS  6
#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 *idp)
{
  int j,i;
  double result=0.0;
  long my_id = (long)idp;
  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) {
      pthread_cond_signal(&count_threshold_cv);
      // ▯▯▯▯pthread_cond_broadcast()▯signal▯▯wait▯thread
      // should use pthread_cond_broadcast()
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n", my_id,
count);
      }
    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 *idp)
{
  long my_id = (long)idp;
```

```
  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("***Before cond_wait: thread %ld\n", my_id);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("***Thread %ld Condition signal received.\n", my_id);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[6];
  pthread_attr_t attr;

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

  /*
  For portability, explicitly create threads in a joinable state
  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[2], &attr, watch_count, (void *)2);
  pthread_create(&threads[3], &attr, watch_count, (void *)3);
  pthread_create(&threads[4], &attr, watch_count, (void *)4);
  pthread_create(&threads[5], &attr, watch_count, (void *)5);
  pthread_create(&threads[0], &attr, inc_count, (void *)0);
  pthread_create(&threads[1], &attr, inc_count, (void *)1);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
```

```
  pthread_exit (NULL);

}
```

bug:

- □□□□pthread_cond_broadcast()□signal□□wait□thread
- should use pthread_cond_broadcast()

3. The bug4.c program is yet another example of what can go wrong when using condition variables. Review, compile (for gcc include the -lm flag) and run the code. Observe the output and then see if you can fix the problem. The explanation is provided in the bug examples table above, and an example solution is provided by the bug4fix.c program.

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

/* Define and scope what needs to be seen by everyone */
#define NUM_THREADS  3
#define ITERATIONS 10
#define THRESHOLD 12
int count = 0;
double finalresult=0.0;
pthread_mutex_t count_mutex;
pthread_cond_t count_condvar;


void *sub1(void *t)
{
  int i;
  long tid = (long)t;

  /* do some work */
  sleep(1);
  /*
  What's wrong here?
  */
  pthread_mutex_lock(&count_mutex);
  if (count < THRESHOLD) {
    printf("sub1: thread=%ld going into wait. count=%d\n",tid,count);
    pthread_cond_wait(&count_condvar, &count_mutex);
    printf("sub1: thread=%ld Condition variable signal received.",tid);
    printf(" count=%d\n",count);
    /* do some work */
    sleep(1);
    count++;
    finalresult += count;
    printf("sub1: thread=%ld count now equals=%d finalresult=%e. Done.\n",
           tid,count,finalresult);
    }
```

```
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

void *sub2(void *t)
{
  int j,i;
  long tid = (long)t;
  double myresult=0.0;

  for (i=0; i<ITERATIONS; i++) {
    for (j=0; j<100000; j++)
      myresult += sin(j) * tan(i);
    pthread_mutex_lock(&count_mutex);
    finalresult += myresult;
    count++;
    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == THRESHOLD) {
      printf("sub2: thread=%ld Threshold reached. count=%d. ",tid,count);
      pthread_cond_signal(&count_condvar);
      printf("Just sent signal.\n");
      }
    else {
      printf("sub2: thread=%ld did work. count=%d\n",tid,count);
      }
    pthread_mutex_unlock(&count_mutex);
    }
    printf("sub2: thread=%ld  myresult=%e. Done. \n",tid,myresult);
  pthread_exit(NULL);
}



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

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

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, sub1, (void *)t1);
  pthread_create(&threads[1], &attr, sub2, (void *)t2);
```

```
  pthread_create(&threads[2], &attr, sub2, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d threads. Final result=%e. Done.\n",
          NUM_THREADS,finalresult);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_condvar);
  pthread_exit (NULL);

}
```

bug:

- note that if THRESHOLD is reached before this waiting thread is run
- pthread_cond_wait will never receive signal,
- thus pthread_cond_wait will never return.

solution:

- A check is made in sub1 to make sure the pthread_cond_wait() call is not made
- if the value of count is not what it expects.
- (add an condition statement: count < COUNT_LIMIT)
- Its work is also placed after it is awakened, while count is locked.