COMS/CPRE 425
Spring 2005
Lecture 17

Ricky A. Kendall
rickyk@cs.iastate.edu
Thread Specific Data (TSD)

- int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *value));
- In general, thread key creation allocates a key that locates data specific to each thread in the process.
- The key, created by pthread_key_create(), is global to all threads in the process, which allows each thread to bind specific value to the key once the key has been created.
- A new key is assigned NULL for all active threads.
- Optionally, a destructor function, "destructor," may be associated with each key.
- Upon thread exit, if a "key" has a non-NULL "destructor" function and the thread has a non-NULL "value" associated with that "key", the "destructor" function is called with the current associated "value."
Setting TSD values

- `int pthread_setspecific(pthread_key_t key, const void *value);`

- Each thread may bind a new value to the key using `pthread_setspecific()`.
  - The values are unique to the binding thread and are individually maintained.
  - These values continue for the life of the calling thread.

- Proper synchronization of key storage and access must be ensured by the caller (e.g., mutexes).

- The value argument to `pthread_setspecific()` is generally a pointer to a block of dynamically allocated memory reserved by the calling thread for its own use.

- You have to cast the real value as a void!!
Getting TSD values

• `void *pthread_getspecific(pthread_key_t key);`

• `pthread_getspecific()` returns the current value bound to the designated key specified by the calling thread.
  ➢ If the key has no value bound to it, the value NULL is returned.

• You have to cast this to the appropriate type!
Destroying TSD keys

- int pthread_key_delete(pthread_key_t key);
- pthread_key_delete() deletes a thread-specific data key formerly created by pthread_key_create().
- The thread-specific data values associated with keys do not have to be NULL.
- It is the application's responsibility to perform cleanup actions related to the deleted key or associated thread-specific data in any threads.
- Cleanup can be done either before or after calling pthread_key_delete(). pthread_key_delete() does not invoke a destructor function.
- Although the "destructor" function should clean-up the key's thread-specific-data storage (e.g., values), pthread_key_delete() needs to be used to free the storage associated with the key.
TSD Example

• For each argument you pass to the executable of this example, a thread is created and privately bound to the string-value of that argument.

```
#include <pthread.h>
void *thread_specific_data(), free();
#define MAX_ARGC 20
pthread_t tid[MAX_ARGC]; int num_threads;
main( int argc, char *argv[] ) {
    int i; num_threads = argc - 1;
    for( i = 0; i < num_threads; i++)
        pthread_create(&tid[i], NULL,
                        thread_specific_data, argv[i+1]);
    for( i = 0; i < num_threads; i++)
        pthread_join(tid[i], NULL);}
/* end main */
```
TSD Example [2]

```c
void *thread_specific_data(char private_data[]) {
    static pthread_mutex_t keylock;
    /* static ensures only one copy of keylock */
    static pthread_key_t key;
    static int once_per_keyname = 0;  void * tsd = NULL;
    if (!once_per_keyname) {
        pthread_mutex_lock(&keylock);
        /* retest with the lock; it may have changed */
        if (!once_per_keyname++)
            pthread_key_create(&key, free);
        pthread_mutex_unlock(&keylock);
    }
}
```
TSD Example [3]

tsd = pthread_getspecific(key);
if (tsd == NULL) {
    tsd = (void *)malloc(strlen(private_data) + 1);
    strcpy(tsd, private_data);
    pthread_setspecific(key, tsd);
    printf("tsd for %d = %s\n",pthread_self(),
            (char *)pthread_getspecific(key));
    sleep(2);
    printf("tsd for %d remains %s\n",pthread_self(),
            (char *)pthread_getspecific(key));
} } /* end thread_specific_data */
Process and thread complexity

- Both mutexes and condition variables
  - Can be shared among threads of a process.
  - Can be shared among threads of multiple processes.

- Consider a thread that
  - Calls exec
    - Replaces the process with the new command
    - Kills all threads of the process
  - Calls fork
    - Creates an exact copy of the process
    - Both processes have the same number of threads.

- There are applications that need to do this but it is very tricky and careful design is necessary.
  - None of the programs we will write will require multiple processes.
Pthreads Examples

• The previous examples;
  ➢ Created a thread for each chunk of work
  ➢ While practical for some things this situation is not practical for a HPC code which:
    ▪ Has large tasks of varied time
    ▪ Uses significant resources for each task

• The matrix multiplication routine shown previously
  ➢ Generated a thread for every element of the product matrix $C_{i,j}$
    ▪ Not practical

• How do we handle the problem?
Limiting the number of threads

• The goal is to provide a user defined maximum number of threads to manage resource utilization.

➢ Even with OpenMP we have to implicitly be concerned with this.
  ▪ On the NERSC systems the 1500,1501,1499 case was unable to run with the default data and stack sizes!
    ♦ Remember -bmaxstack and -bmaxdata ?

➢ You have to balance the granularity manually.
  ▪ Essentially the “work” load per task

➢ By limiting the number of threads you also minimize the parallel overhead.
Governing Thread Count

• There are many ways to govern the number of threads.

  ➢ Generate a pool of threads
    ▪ All working threads are created as a part of the pool and wait for work to be assigned.
      ♦ Essentially a queue data structure
    ▪ Work is dynamically assigned to them.
    ▪ Once the work is completed then the thread returns to the pool and is available for further tasks.
    ▪ Fully dynamic mechanism.

  ➢ Generate a fixed number of threads and loop work assignments to them.
    ▪ Instead of a queue this is done using a stack based mechanism.
    ▪ Generate the threads with work assigned.
    ▪ Wait for the “stacked” thread to complete
    ▪ Assign the NEXT task to the thread.
    ▪ Not a fully dynamic mechanism.
Stack Example.
How the pool works

init

Add Work

Collect results

destroy
The pool tool

- Handout Take one and pass it along.
The makefile.

CC = gcc
CFLAGS = -O
THREAD_CFLAGS = \${CFLAGS} -lpthread
POOL_OBJS = tpool_init.o tpool_thread.o
tpool_destroy.o tpool_add_work.o

tpooltest: tpooltest.o \$(POOL_OBJS)
 \${CC} \${THREAD_CFLAGS} tpooltest.o
 \$(POOL_OBJS) -o tpooltest

clean :
 rm -f *.o *~ *# core tpooltest
The thread code

```c
#include <stdio.h>
#include <pthread.h>
#include "tpool.h"

void r1(char * printstring) {
    int i, x;
    printf("%s START\n", printstring);
    for (i = 0; i < 1000000; i++) {
        x = x +i;
    }
    printf("%s DONE\n", printstring);
}
```
The main testing code

```c
#include <stdio.h>
#include <pthread.h>
#include "tpool.h"


main(void)
{
    extern char *s1[];  int i;  pthread_attr_t attributes;  tpool_t test_pool;
    tpool_init(&test_pool, 8, 20, 0);  printf("main: initialization done\n");
    sleep(5);  /* let initialization catch up*/
    for ( i = 0; i < 20; i++) tpool_add_work(test_pool, r1, s1[i]));
    printf("main: all work queued\n");
    tpool_destroy(test_pool, 1);
    return 0;}
```
The pool include file

typedef struct tpool_work {
    void (*routine)();
    void *arg;
    struct tpool_work *next;
} tpool_work_t;

typedef struct tpool {
    /* pool characteristics */
    int num_threads; int max_queue_size; int do_not_block_when_full;
    pthread_t *threads;
    int cur_queue_size;
    tpool_work_t *queue_head;
    tpool_work_t *queue_tail;
    int queue_closed;
    int shutdown;
    /* pool synchronization */
    pthread_mutex_t queue_lock;
    pthread_cond_t queue_not_empty;
    pthread_cond_t queue_not_full;
    pthread_cond_t queue_empty;
} *tpool_t;
The pool initialization

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include "tpool.h"

void *tpool_thread(void *);

void tpool_init(tpool_t *tpoolp,
               int num_worker_threads,
               int max_queue_size,
               int do_not_block_when_full)
{

The pool initialization[2]

```c
int i, rtn;
tpool_t tpool;
/* allocate a pool data structure */
if ((tpool = (tpool_t )malloc(sizeof(struct tpool))) == NULL)
    perror("malloc"), exit(1);
/* initialize th fields */
tpool->num_threads = num_worker_threads;
tpool->max_queue_size = max_queue_size;
tpool->do_not_block_when_full = do_not_block_when_full;
if ((tpool->threads =
    (pthread_t *)malloc(sizeof(pthread_t)*num_worker_threads))
    == NULL)
    perror("malloc"), exit(1);
tpool->cur_queue_size = 0;
tpool->queue_head = NULL; tpool->queue_tail = NULL;
tpool->queue_closed = 0; tpool->shutdown = 0;
```
The pool initialization[3]

if ((rtn = pthread_mutex_init(&(tpool->queue_lock), NULL)) != 0)
    fprintf(stderr,"pthread_mutex_init %s",strerror(rtn)), exit(1);
if ((rtn = pthread_cond_init(&(tpool->queue_not_empty), NULL)) != 0)
    fprintf(stderr,"pthread_cond_init %s",strerror(rtn)), exit(1);
if ((rtn = pthread_cond_init(&(tpool->queue_not_full), NULL)) != 0)
    fprintf(stderr,"pthread_cond_init %s",strerror(rtn)), exit(1);
if ((rtn = pthread_cond_init(&(tpool->queue_empty), NULL)) != 0)
    fprintf(stderr,"pthread_cond_init %s",strerror(rtn)), exit(1);
if (pthread_attr_init(&attributes))
    fprintf(stderr,"main: attribute init failed\n");
if (pthread_attr_setscope(&attributes,PTHREAD_SCOPE_SYSTEM))
    fprintf(stderr,"main: attribute set scope SYSTEM failed\n");

/* create threads */
for (i = 0; i != num_worker_threads; i++) {
    if ((rtn = pthread_create( &tpool->threads[i],
                                attributes, tpool_thread, (void *)tpool)) != 0)
        fprintf(stderr,"pthread_create %d",rtn), exit(1);
}  *tpoolp = tpool;
Adding work to the queue/pool

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include "tpool.h"

int tpool_add_work(tpool_t tpool, void (*routine)(), void *arg) {
    int rtn; tpool_work_t *workp;
    if ((rtn = pthread_mutex_lock(&(tpool->queue_lock))) != 0)
        fprintf(stderr,"pthread_mutex_lock %d",rtn), exit(1);
    /* no space and this caller doesn't want to wait */
    if ((tpool->cur_queue_size == tpool->max_queue_size) &&
        tpool->do_not_block_when_full) {
        if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
            fprintf(stderr,"pthread_mutex_unlock %d",rtn), exit(1);
        return -1;
    }
}
```
Adding work to the queue/pool[2]

while( (tpool->cur_queue_size == tpool->max_queue_size) &&
       (!(tpool->shutdown || tpool->queue_closed)) ) {
  if ((rtn = pthread_cond_wait(&(tpool->queue_not_full),
                               &(tpool->queue_lock))) != 0)
    fprintf(stderr,"pthread_cond_wait %d",rtn), exit(1);
}
/* the pool is in the process of being destroyed */
if (tpool->shutdown || tpool->queue_closed) {
  if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
    fprintf(stderr,"pthread_mutex_unlock %d",rtn), exit(1);
  return -1; }
/* allocate work structure */
if (((workp = (tpool_work_t *)malloc(sizeof(tpool_work_t))) == NULL)
    perror("malloc"), exit(1);
  workp->routine = routine;
  workp->arg = arg;
  workp->next = NULL;
  printf("adder: adding an item %d\n", workp->routine);
Adding work to the queue/pool[3]

if (tpool->cur_queue_size == 0) {
    tpool->queue_tail = tpool->queue_head = workp;

    printf("adder: queue == 0, waking all workers\n");

    if ((rtn = pthread_cond_broadcast(&(tpool->queue_not_empty))) != 0)
        fprintf(stderr,"pthread_cond_signal %d",rtn), exit(1);;
} else {
    tpool->queue_tail->next = workp;
    tpool->queue_tail = workp;
}

    tpool->cur_queue_size++;
    if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
        fprintf(stderr,"pthread_mutex_unlock %d",rtn), exit(1);
    return 1;
}
Doing the thread work (dynamic)

```c
void *tpool_thread(void *arg)
{
    tpool_t tpool = (tpool_t)arg;
    int rtn;  tpool_work_t *my_workp;
    for(;;) {
        /* Check queue for work */
        if (((rtn = pthread_mutex_lock(&(tpool->queue_lock))) != 0)
            fprintf(stderr,"pthread_mutex_lock %d",rtn), exit(1);
            while (((tpool->cur_queue_size == 0) && (!tpool->shutdown))
            printf("worker %d: I'm sleeping again\n", pthread_self());
            if (((rtn = pthread_cond_wait(&(tpool->queue_not_empty),
                &(tpool->queue_lock))) != 0)
                fprintf(stderr,"pthread_cond_wait %d",rtn), exit(1);
        }  sleep(5);
        printf("worker %d: I'm awake\n", pthread_self());
    }
```
Doing the thread work [2]

/* Has a shutdown started while i was sleeping? */
if (tpool->shutdown == 1) {
    if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
        fprintf(stderr,"pthread_mutex_unlock \%d",rtn), exit(1);
    pthread_exit(NULL);
}

/* Get to work, dequeue the next item */
my_workp = tpool->queue_head;
tpool->cur_queue_size--;  
if (tpool->cur_queue_size == 0)
    tpool->queue_head = tpool->queue_tail = NULL;
else
    tpool->queue_head = my_workp->next;
printf("worker \%d: dequeing item \%d\n", pthread_self(), my_workp->next);
Doing the thread work [3]

/* Handle waiting add_work threads */
if (!tpool->do_not_block_when_full) &&
  (tpool->cur_queue_size == (tpool->max_queue_size - 1))
  if ((rtn = pthread_cond_broadcast(&(tpool->queue_not_full))) != 0)
    fprintf(stderr,"pthread_cond_broadcast %d",rtn), exit(1);

/* Handle waiting destroyer threads */
if (tpool->cur_queue_size == 0)
  if ((rtn = pthread_cond_signal(&(tpool->queue_empty))) != 0)
    fprintf(stderr,"pthread_cond_signal %d",rtn), exit(1);

if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
  fprintf(stderr,"pthread_mutex_unlock %d",rtn), exit(1);

/* Do this work item */
(*(my_workp->routine))(my_workp->arg);
free(my_workp);
} return(NULL);   }
Destroy the pool

```c
int tpool_destroy(tpool_t tpool, int finish)
{
    int i,rtn;
    tpool_work_t *cur_noddep;

    if ((rtn = pthread_mutex_lock(&(tpool->queue_lock))) != 0)
        fprintf(stderr,"pthread_mutex_lock %d",rtn), exit(1);

    /* Is a shutdown already in progress? */
    if (tpool->queue_closed || tpool->shutdown) {
        if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
            fprintf(stderr,"pthread_mutex_unlock %d",rtn), exit(1);
        return 0;
    }

    tpool->queue_closed = 1;
}
```
Destroy the pool [2]

/* If the finish flag is set, wait for workers to drain queue */
if (finish == 1) {
    while (tpool->cur_queue_size != 0) {
        if ((rtn = pthread_cond_wait(&(tpool->queue_empty),
                                     &(tpool->queue_lock))) != 0)
            fprintf(stderr, "pthread_cond_wait %d", rtn), exit(1);
    }
}

tpool->shutdown = 1;

if ((rtn = pthread_mutex_unlock(&(tpool->queue_lock))) != 0)
    fprintf(stderr, "pthread_mutex_unlock %d", rtn), exit(1);
Destroy the pool [3]

/* Wake up any workers so they recheck shutdown flag */
if ((rtn = pthread_cond_broadcast(&(tpool->queue_not_empty))) != 0)
    fprintf(stderr,"pthread_cond_broadcast %d",rtn), exit(1);
if ((rtn = pthread_cond_broadcast(&(tpool->queue_not_full))) != 0)
    fprintf(stderr,"pthread_cond_broadcast %d",rtn), exit(1);

/* Wait for workers to exit */
for(i=0; i < tpool->num_threads; i++) {
    if ((rtn = pthread_join(tpool->threads[i],NULL)) != 0)
        fprintf(stderr,"pthread_join %d",rtn), exit(1);
}
/* Now free pool structures */
free(tpool->threads);
while(tpool->queue_head != NULL) {
    cur_nodep = tpool->queue_head->next;
    tpool->queue_head = tpool->queue_head->next;
    free(cur_nodep);
}
free(tpool); }
Output

worker 1026: I'm sleeping again
worker 2051: I'm sleeping again
worker 3076: I'm sleeping again
worker 4101: I'm sleeping again
worker 5126: I'm sleeping again
worker 6151: I'm sleeping again
worker 7176: I'm sleeping again
worker 8201: I'm sleeping again
main: initialization done
adder: adding an item 134515024
adder: queue == 0, waking all workers
tpool_add_work returned 1
adder: adding an item 134515024
tpool_add_work returned 1
adder: adding an item 134515024
tpool_add_work returned 1
adder: adding an item 134515024
tpool_add_work returned 1
adder: adding an item 134515024
Output [2]

tpool_add_work returned 1
adder: adding an item 134515024
...
main: all work queued
worker 5126: I'm awake
worker 5126: dequeing item 134531344
STRING 0 START
STRING 0 DONE
worker 4101: I'm awake
worker 4101: dequeing item 134531360
STRING 1 START
STRING 1 DONE
worker 3076: I'm awake
worker 3076: dequeing item 134531376
STRING 2 START
STRING 2 DONE
worker 2051: I'm awake
worker 2051: dequeing item 134531392
STRING 3 START
STRING 3 DONE
Output [3]

worker 1026: I'm awake
worker 1026: dequeing item 134531408
STRING 4 START
STRING 4 DONE
...
worker 2051: I'm awake
worker 2051: dequeing item 0
STRING 19 START
STRING 19 DONE
worker 1026: I'm sleeping again
worker 7176: I'm sleeping again
worker 5126: I'm sleeping again
worker 3076: I'm sleeping again
worker 2051: I'm awake
worker 3076: I'm awake
worker 4101: I'm awake
worker 5126: I'm awake
worker 6151: I'm awake
worker 8201: I'm awake
worker 8201: I'm sleeping again
worker 6151: I'm sleeping again
worker 4101: I'm sleeping again
worker 2051: I'm sleeping again
worker 2051: I'm awake
worker 3076: I'm awake
worker 4101: I'm awake
worker 5126: I'm awake
worker 6151: I'm awake
worker 7176: I'm awake
worker 1026: I'm awake
Patches and such.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0(14)</td>
<td>1(15)</td>
<td>2(16)</td>
<td></td>
</tr>
<tr>
<td>3(20)</td>
<td>4(21)</td>
<td>5(22)</td>
<td></td>
</tr>
<tr>
<td>6(26)</td>
<td>7(27)</td>
<td>8(28)</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0(14)</td>
<td>1(15)</td>
<td>2(16)</td>
<td></td>
</tr>
<tr>
<td>3(17)</td>
<td>4(18)</td>
<td>5(19)</td>
<td></td>
</tr>
<tr>
<td>6(20)</td>
<td>7(21)</td>
<td>8(22)</td>
<td></td>
</tr>
</tbody>
</table>
Copy_patch

/*-- function to copy a patch out of a matrix   ---------------*/
void copy_patch(
    double *source,     /* pointer to source array containing patch */
    int src_row_dim,    /* row dimension of source array */
    int src_col_dim,    /* column dimension of source array */
    int src_row_index,  /* row index of first element of the patch in the source 
                        array */
    int src_col_index,  /* column index of first element of the patch in the source 
                        array */
    int num_src_rows,   /* Number of rows in the patch */
    int num_src_cols,   /* Number of columns in the patch */
    double *target,     /* pointer to the target array */
    int trg_row_dim,    /* row dimension of the target array */
    int trg_col_dim,    /* column dimension of the target array */
    int trg_row_index,  /* row index of first element of the patch in the target 
                        array */
    int trg_col_index   /* column index of first element of the patch in the target 
                        array */
)

Copy_patch [2]

{
    int i,j;
    for (i=0;i<\text{num\_src\_rows};i++)
        for (j=0;j<\text{num\_src\_cols};j++) {
            \text{target}[\text{PANEL}((i+\text{trg\_row\_index}),
                (j+\text{trg\_col\_index}),
                \text{trg\_row\_dim,} \text{trg\_col\_dim})]

            =

            \text{source}[\text{PANEL}((i+\text{src\_row\_index}),
                (j+\text{src\_col\_index}),
                \text{src\_row\_dim,} \text{src\_col\_dim})];
        }
}
}
Accumulate_patch

void accumulate_patch(
    double *source,    /* pointer to source array containing patch */
    int src_row_dim,   /* row dimension of source array */
    int src_col_dim,   /* column dimension of source array */
    int src_row_index, /* row index of first element of the patch in the source 
                        array */
    int src_col_index, /* column index of first element of the patch in the source 
                        array */
    int num_src_rows,  /* Number of rows in the patch */
    int num_src_cols,  /* Number of columns in the patch */
    double *target,    /* pointer to the target array */
    int trg_row_dim,   /* row dimension of the target array */
    int trg_col_dim,   /* column dimension of the target array */
    int trg_row_index, /* row index of first element of the patch in the target array */
    int trg_col_index  /* column index of first element of the patch in the target array */
)

Accumulate_patch [2]

```c
{
    int i,j;
    for (i=0;i<num_src_rows;i++) {
        for (j=0;j<num_src_cols;j++) {
            target[PANEL((i+trg_row_index),
                        (j+trg_col_index),
                        trg_row_dim,trg_col_dim)]
                +=

                source[PANEL((i+src_row_index),
                            (j+src_col_index)
                            ,src_row_dim,src_col_dim)];
        }
    }
}
```