ID1217 Concurrent Programming
Lecture 6

ROYAL INSTITUTE OF TECHNOLOGY

# Tutorial: Introduction to Pthreads

Vladimir Vlassov
KTH/ICT/EECS

# Additional reading

- Tutorial: POSIX Threads Programming at https://computing.llnl.gov/tutorials/pthreads/
- Ch 4 in An Introduction to Parallel Programming, by Peter Pacheco, Morgan Kaufmann (available online via KTH library)

# Outline

- Pthreads – IEEE POSIX threads API
  - Pthreads types and functions
  - Thread creation and termination
  - Joinable and detached threads
  - Synchronization primitives: Mutex locks, Condition variables, Semaphores
  - Examples

ROYAL INSTITUTE OF TECHNOLOGY

# Pthreads: POSIX Threads

- **Pthreads** is a standard set of C library routines for multithreaded programming with shared variables
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Allows to create and synchronize multiple threads in a heavy-weight process
  - Threads share a common address space (thru common variables)
  - Each thread has a private stack for local variables
- Goal in developing the Pthreads API:
  - “ To give programmers the ability to write concurrent applications that run on both uniprocessor and multiprocessor machines transparently, taking advantage of the additional processors if any.”

ROYAL INSTITUTE OF TECHNOLOGY

# Pthreads API

- The Pthreads API includes a library of functions that contains in total
  - over 60 functions excluding extensions;
  - about 100 subroutines including extensions.
- Thread management (29 functions): create, exit, detach, join, get/set attributes, …
- Thread cancellation (9): cancel, test for cancellation, ...
- Mutex locks (19): init, destroy, lock, unlock, try lock, get/set attributes;
- Condition variables (11): init, destroy, wait, timed wait, signal, broadcast, get/set attributes;
- Read/Write locks (13): init, destroy, write lock/unlock, read lock/unlock, get/set attributes;
- Thread specific storage (4)
- Signals (3): send a signal to thread, signal mask
- Unsupported extension (16): get/set scheduling policy, …
- Extension: semaphores (semaphore.h)

# Header Files, Compiling and Linking

- Header files:

  `#include <pthread.h>`

  `#include <sched.h>`

  `#include <semaphore.h>` to use semaphores

- Compile using **gcc** (or **g++**) and link to **pthreadlib**
  - For example:

    ```
    g++ -c -O3 common.cpp
    g++ –O3 -o myprog myprog.cpp common.o -lm –lpthread
    ```

  - To link with POSIX 4 extension (e.g. semaphores): `-lposix4`

- Run as an ordinary executable code:

  `.\myprog <command line parameters>`

# Pthreads Naming Convention

- Types: **`pthread`**[_*object*][**_np**]**`_t`**
- Functions: **`pthread`**[_*object*]**_action**[**_np**]
- Constants and Macros: **`PTHREAD_`***PURPOSE*[**_NP**]

- If the type of an object is not a thread, then:
  - ***object*** represents the type of object, e.g., **`mutex`**
  - **action** is an operation to be performed on the object, e.g., **`lock`**
  - **np** or **NP** indicates that the name or symbol is a non-portable extension to the API set,
  - ***PURPOSE*** indicates the use or purpose of the symbol, e.g. **`PTHREAD_CREATE_DETACHED`**

# Primitive Data Types

| type | Description |
|---|---|
| pthread_attr_t | Thread creation attribute |
| pthread_cleanup_entry_np_t | Cancelation cleanup handler entry |
| pthread_condattr_t | Condition variable creation attribute |
| pthread_cond_t | Condition Variable synchronization primitive |
| pthread_joinoption_np_t | Options structure for extensions to pthread_join() |
| pthread_key_t | Thread local storage key |
| pthread_mutexattr_t | Mutex creation attribute |
| pthread_mutex_t | Mutex (Mutual exclusion) synchronization primitive |
| pthread_once_t | Once time initialization control variable |
| pthread_option_np_t | Pthread run-time options structure |
| pthread_rwlockattr_t | Read/Write lock attribute |
| pthread_rwlock_t | Read/Write synchronization primitive |
| pthread_t | Pthread handle |
| pthread_id_np_t | Thread ID. For use as an integral type. |
| struct sched_param | Scheduling parameters (priority and policy) |

# Thread Declaration and Creation

Declaration:

```
pthread_t tid; /* thread handle */
```

Creation:

```
int pthread_create(pthread_t* tid,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void *),
                   void *arg);
```

- Analogous to a combined **fork** and **exec** routine
- The routine creates a thread with the specified thread attributes.
- When **attr** is **NULL** the default thread attributes are used.
- Returns a thread id in **tid** .
- The new thread begins execution by calling **start_routine** with a single argument **arg**

- For example:

```
pthread_create (&tid, &tattr,(void *) tf, (void *) &x);
```

# Thread Attributes

- Thread attributes can be set before a thread is created
  - If not set, the attributes are set to default values
- Typical usage:
  1. Declare a pthread attribute variable of the **`pthread_attr_t`** data type
  2. Initialize the attribute variable:
     **`pthread_attr_init(&attr)`**
  3. Set an attribute value:
     **`pthread_attr_set`*attribute*`(&attr, value)`**
  4. Destroy the attribute var if not longer needed:
     **`pthread_attr_destroy(&attr)`**

# Thread Attributes (cont'd)

- Detachable or joinable thread
  - **`detachstate`** – indicates whether a thread is detached or joinable.
- Stack information
  - **`stackaddr`** – specifies the stack address (void pointer) of a thread created with this attributes object.
  - **`stacksize`** – specifies the minimum stack size, in bytes, of a thread created with this attributes object.
- Thread scheduling attributes
  - **`schedparam`** – specifies the scheduling parameters such as priority.
  - **`schedpolicy`** – specifies the scheduling policy (FIFO, round-robin, other).
  - **`inheritsched`** – specifies the inheritance of scheduling properties (either inherits or not).
  - **`scope`** – the contention-scope attribute specifies the contention scope (local or global) of a thread

# Passing Arguments to a Thread

- **`pthread_create`** permits to pass one argument (a void pointer) to the thread start routine.
  - The argument must be passed by reference and cast to **`void *`**.
  - To pass multiple arguments, collect them into a structure and pass a pointer to that structure.
  - Important: the argument data structure is located in the parent thread's memory space and it must not be corrupted/modified until the thread has finished accessing it.

# Thread Identifiers

- When created, a thread is assigned a unique thread identifier (handle) that is used to reference the thread

```
pthread_t tid;
…
pthread_create(&tid, …)
…
pthread_join(tid, NULL);
```

- A thread can get its own handle:

```
pthread_t mytid = pthread_self()
```

  - returns the handle of the calling thread
- To compare two threads, use

```
pthread_equal(tid1, tid2)
```

  - returns 0 if different
  - Should not use == to compare threads.

ROYAL INSTITUTE OF TECHNOLOGY

# Thread Termination

A thread may terminate its execution in two ways

- Explicitly by executing

```
void pthread_exit(void* return_value);
```

  - `return_value` – a single return value or `NULL`
  - Analogous to `exit`
  - If the current thread is the last thread then the process terminates
  - The `exit` routine kills all threads and exits the process

- Implicitly by returning from the thread routine

```
return (status)
```

  - Return from the thread routine is equivalent to calling pthread_exit
  - Return from the initial thread main is the equivalent to calling exit

# Join a Thread

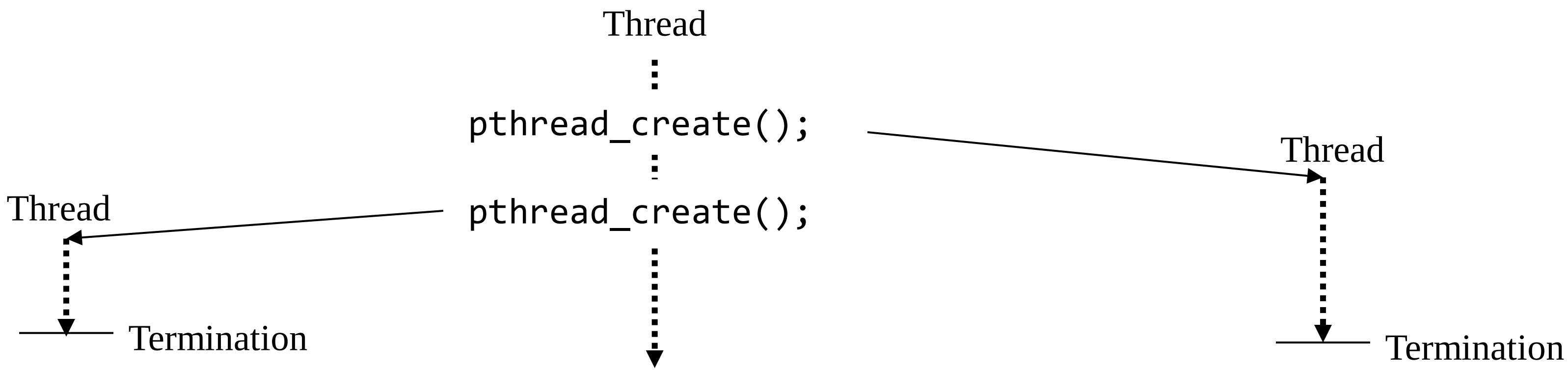
- A thread can wait for its child thread to terminate, i.e. it joins the child thread by executing:

```
int pthread_join(pthread_t tid, void** value);
```

  - `tid` – child's descriptor
  - `value` –the address of the return value
  - Analogous to the UNIX **`wait`**
  - Must specify thread. There is no wait any.
  - Current thread blocks until the child thread terminates
  - The return value of thread is returned in **`value`**
- All threads can be either detached or joinable.

# Detached Threads

- Threads that are not joinable are called **detached threads**.
  - A parent does not need to join a child.
  - When a detached thread terminates, it is destroyed.




- A thread can be created as "detached"

```
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED)
pthread_create(&tid, &tattr, …)
```

- A thread can be "detached" explicitly by calling

```
int pthread_detach(pthread_t tid);
```

# Example 1: Independent Threads

- Creates a number of threads and then joins them.
- Each thread estimates its life time.
- The main thread waits for the child threads to terminate and then prints the total life time.

**ind.c** on the course website under "Lectures"

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/times.h>
#define MAXTHREADS 8
void *tf (void *);
int n = MAXTHREADS, results[MAXTHREADS], total = 0;
int main (int argc, char *argv[]) {
  pthread_t tid[MAXTHREADS];
  long i;
  n = (argc > 1)? atoi(argv[1]) : MAXTHREADS;
  for (i = 0; i < n; i++) pthread_create (&tid[i], NULL, tf, (void *) i);
  for (i = 0; i < n; i++) pthread_join (tid[i], NULL); total +=
    results[i];
  printf ("Total: %d\n", total);
  return 1;
}
void *tf (void *num) {
  int i;
  long mynum = (long)num;
  struct tms buffer;
  clock_t time = times (&buffer);
  for (i = 0; i < 5; i++)
     printf ("I am thread %d of %d\n", mynum, n); sleep (rand() % 2);
  time = times (&buffer) - time;
  results[mynum] = (int) time;
}
```

# Synchronization Mechanisms in Pthreads

- **Mutexs** – mutual exclusion locks
- **Condition variables** – queues of thread (used for condition synchronization)
- **Reader/Writer locks** – shared read and exclusive write locks
- **Semaphores** – an extension (POSIX 1b) to Pthreads API implemented on top of mutexes and condition variables:

```
#include <semaphore.h>
```

Link to `-lposix4`

# Mutexs

- **Mutex** is an object of the **`pthread_mutex_t`** type used to protect critical sections of code, i.e. for mutual exclusion
  - Acts like a lock protecting access to shared data within a critical section: Lock, unlock and trylock operations
  - Before to be used, must be initialized.
  - A thread can set/get mutex attributes.
  - Should be destroyed when no longer needed.
- Declaration:

```
pthread_mutex_t mutex;
```

ROYAL INSTITUTE OF TECHNOLOGY

# Functions on Mutex

- `pthread_mutex_init(&mutex, &attr)`
  - Create and initialize a new mutex object, set its attributes.
  - The mutex is initially unlocked.
- `pthread_mutex_destroy(mutex)`
  - Destroy the mutex when no longer needed
- `pthread_mutex_lock(&mutex)`
  - Lock the mutex. If it is already locked, the call blocks the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock(&mutex)`
  - Attempt to lock a mutex. Returns non-zero value if the mutex is already locked, otherwise returns 0.
- `pthread_mutex_unlock(&mutex)`
  - Unlock the mutex (if called by the owning thread)

# Typical Usage

- Create and initialize mutex, …, spawn threads, …, lock the mutex, execute critical section, unlock the mutex, … destroy the mutex
- For example:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
…
pthread_mutex_lock(&mutex);
critical section;
pthread_mutex_unlock(&mutex);
non-critical section;
```

**counters.c** on the course website under “Lectures”

# Example 2: Counters

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5           /* default number of threads */
/* shared variables */
double total;
pthread_mutex_t lock;
void *Counter (void *null) {
  int i;
  double *result = (double *) calloc (1, sizeof (double));
  for (i = 0; i < 1000000; i++) *result = *result + (double) (random ()%100);
  pthread_mutex_lock (&lock);
  total += *result;
  pthread_mutex_unlock (&lock);
  pthread_exit ((void *) result);
}
int main (int argc, char *argv[]) {
  int n = NUM_THREADS;
  double *result;
  result = (double*)calloc(1, sizeof(double));
  pthread_t thread[NUM_THREADS];
  int t;
  if (argc > 1) n = atoi (argv[1]);
  if (n > NUM_THREADS || n < 1) n = NUM_THREADS;
  for (t = 0; t < n; t++) pthread_create (&thread[t], NULL, Counter, NULL);
  for (t = 0; t < n; t++) {
      pthread_join (thread[t], (void*) &result);
      printf ("Completed join with thread %d. Result =%f\n", t, *result);
    }
  printf ("The total = %f\n", total);
}
```

# Condition Variables

- **A condition variable (a.k.a. queue variable)** is an object of the **`pthread_cond_t`** data type used for blocking and resuming threads holding mutexs
  - Used for condition synchronization.
  - A condition variable is always used in conjunction with a mutex lock.
- Declaration

```
pthread_cond_t cond;
```

- Operations on condition variables:
  - wait, signal, broadcast
  - Signal-and-continue signaling discipline

# Functions on Condition Variables

- `pthread_cond_init(&cond, &attr)`
  - Create and initialize a new condition variable.
- `pthread_cond_destroy(&cond)`
  - Free the condition variable that is no longer needed.
- `pthread_cond_wait(&cond, &mutex)`
  - Wait on the condition variable, i.e. release the mutex (if owner) and place the calling thread to the tail on the cond var queue.
- `pthread_cond_signal(&cond)`
  - Signal the condition variable, i.e. move a waiting thread (if any) from the head of the cond var queue to the mutex queue.
- `pthread_cond_broadcast(&cond)`
  - Signal all: Awaken all threads waiting on the condition variable.

# Typical Usage of A Condition Variable

- Take an action when **x == 0**

```
<await (x==0) take_action();>

action() {
  …
  pthread_mutex_lock(&m);
  while (x != 0)
    pthread_cond_wait(&cv, &m);
  take_action();
  pthread_mutex_unlock(&m);
  …
}
```

```
counter() {
  …
  pthread_mutex_lock(&m);
  x--;
  if (x == 0)
    pthead_cond_signal(&cv);
  pthread_mutex_unlock(&m);
  …
}
```

# Example 3: Summing Matrix Elements

**matrixSum.c** on the course website under “Lectures”

```
#define _REENTRANT
#endif
#include <pthread.h>
#include <stdio.h>
#define SHARED 0
#define MAXSIZE 10000  /* maximum matrix size */
#define MAXWORKERS 8   /* maximum number of workers */
pthread_mutex_t barrier;  /* mutex lock for the barrier */
pthread_cond_t go;        /* condition variable for leaving */
int numWorkers;
int numArrived = 0;       /* number who have arrived */
void Barrier() {  /* a reusable counter barrier */
  pthread_mutex_lock(&barrier);
  numArrived++;
  if (numArrived == numWorkers) {
    numArrived = 0;
    pthread_cond_broadcast(&go);
  } else pthread_cond_wait(&go, &barrier);
  pthread_mutex_unlock(&barrier);
}
int size, stripSize;  /* assume size is multiple of numWorkers */
int sums[MAXWORKERS];
int matrix[MAXSIZE][MAXSIZE];
void *Worker(void *);
```

# Summing Matrix Elements (cont'd)

```
/* read command line, initialize, and create threads */
int main( int argc, char *argv[] ) {
  int i, j;
  pthread_attr_t attr;
  pthread_t workerid[MAXWORKERS];
  /* set global thread attributes */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  /* initialize mutex and condition variable */
  pthread_mutex_init(&barrier, NULL);
  pthread_cond_init(&go, NULL);
  /* read command line args if any */
  size = (argc > 1)? atoi(argv[1]) : MAXSIZE;
  numWorkers = (argc > 2)? atoi(argv[2]) : MAXWORKERS;
  if (size > MAXSIZE) size = MAXSIZE;
  if (numWorkers > MAXWORKERS) numWorkers = MAXWORKERS;
  stripSize = size/numWorkers;
  /* initialize the matrix */
  for (i = 0; i < size; i++) {
           printf("[ ");
           for (j = 0; j < size; j++) {
               matrix[i][j] = rand()%99;
               printf(" %d", matrix[i][j]);
           }
           printf(" ]\n");
  }
  /* create the workers, then exit */
  for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr, Worker, (void *) i);
  pthread_exit(NULL);
}
```

ROYAL INSTITUTE OF TECHNOLOGY

# Summing Matrix Elements (cont’d)

```
/* Each worker sums the values in one strip of the matrix.
   After a barrier, worker(0) computes and prints the total */
void *Worker(void *arg) {
  int myid = (int) arg;
  int total, i, j, first, last;

  printf("worker %d (pthread id %d) has started\n",
         myid, pthread_self());

  /* determine first and last rows of my strip */
  first = myid * stripSize;
  last = (myid == numWorkers - 1) ? (size - 1) :
                                    (first + stripSize - 1);

  /* sum values in my strip */
  total = 0;
  for (i = first; i <= last; i++)
    for (j = 0; j < size; j++)
      total += matrix[i][j];
  sums[myid] = total;
  Barrier();
  if (myid == 0) {
    total = 0;
    for (i = 0; i < numWorkers; i++)
      total += sums[i];
    printf("the total is %d\n", total);
  }
}
```

# Thread-Safe (Reentrant) Functions

- A multithreaded program must use **thread-safe (reentrant)** versions of standard functions.
  - A **re-entrant function** can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other.
  - A **thread-safe function** is either re-entrant or protected from multiple simultaneous execution by some form of mutual exclusion.
- To get declared reentrant (thread safe) versions of standard functions, define the `_REENTRANT` macro before any **`include`**:

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <stdio.h>
#include <pthread.h>
...
```

# Semaphore Extension

- A pthread **semaphore** is a special kind of integer object of the **`sem_t`** type that can take any nonnegative value and can be altered by **`sem_wait`** (P, decrement) and **`sem_post`** (V, increment) operations
  - If the semaphore is zero, the P operation blocks the calling thread until the semaphore is positive.

- **`#include <semaphore.h>`**
  - Defines semaphore types and functions

# Semaphores (cont'd)

- Declaration: `sem_t sem;`
- Functions
  - `set_wait(&sem)` – P(sem) – decrements the semaphore if it's positive, otherwise blocks the process until the semaphore is positive
  - `sem_post(sem)` – V(sem) – increments the semaphore
- For example:

```
sem_init(&sem, SHARED, 1);/* sem = 1; */
…
sem_wait(&sem);            /* P(sem) */
Critical section;
sem_post(&sem);            /* V(sem) */
```

ROYAL INSTITUTE OF TECHNOLOGY

# Example 4: Simple Producer/Consumer Using Semaphores

**pc.sems.c** on the course website under “Lectures”

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1
void *Producer(void *);  /* the two threads */
void *Consumer(void *);
sem_t empty, full;    /* the global semaphores */
int data;             /* shared buffer         */
int numIters;
/* main() -- read command line and create threads, then
             print result when the threads have quit */
int main( int argc, char *argv[] ) {
  /* thread ids and attributes */
  pthread_t pid, cid;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  numIters = atoi(argv[1]);
  sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  sem_init(&full, SHARED, 0);   /* sem full = 0  */
  printf("main started\n");
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
  printf("main done\n");
}
```

# Example4: Simple Producer/Consumer Using Semaphores (cont'd)

```
/* deposit 1, ..., numIters into the data buffer */
void *Producer( void *arg ) {
  int produced;
  printf("Producer created\n");
  for (produced = 0; produced < numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
  }
}
/* fetch numIters items from the buffer and sum them */
void *Consumer( void *arg ) {
  int total = 0, consumed;
  printf("Consumer created\n");
  for (consumed = 0; consumed < numIters; consumed++) {
    sem_wait(&full);
    total = total+data;
    sem_post(&empty);
  }
  printf("for %d iterations, the total is %d\n", numIters, total);
}
```