An introduction to the Posix Thread API

- General introduction
- Creation and termination
- Mutex locks
- An example: parallel sum or inner product
**Threads**

- Mode of programming for shared memory [shared address space or symmetric multi-processing (SMP)]
- Very common – supported by all vendors. Part of unix standard.
- Low-level
- Helps understand issues with racing, synchronization, etc.
- Here: we will provide basic overview + cover an example.

**Pros:** simple approach –

**Cons:** Limited to SMPs. Gets complicated for longer codes
The basic commands

See https://computing.llnl.gov/tutorials/pthreads/ among many resources for details.

“a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.” (source: above)

- Initially, main() comprises a single thread
- Programmer can instruct the program to start threads that execute independently.
- *However*: you are responsible for coordinating the concurrent accesses/ modifications of memory variables by different threads
- Once initiated a thread can itself create other threads
Thread creation

\texttt{pthread\_create (thread, attr, thread\_fun, arg)}

\begin{itemize}
  \item \texttt{thread} is of type \texttt{pthread\_t = (unique) identifier of thread}
  \item \texttt{attr} is of type \texttt{pthread\_attr\_t = may be used to set thread attributes.}
  \item \texttt{thread\_fun} is a function pointer. The thread will execute this function after creation.
  \item \texttt{arg} is of type \texttt{*void}. This argument is passed to the function \texttt{thread\_fun}. If more than one argument use a struct
\end{itemize}
`pthread_join`: blocks calling thread until specified thread ends. Allows to synchronize

`pthread_join (threadid, status)`

`threadid` is of type `pthread_t = identifier of thread`

`status` is of type `void**`. 
Use the attribute for declaring thread as *joinable*

```c
pthread_attr_t attr; // declare
pthread_attr_init(&attr); // initialize
pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_JOINABLE); // set
......
/* at end */
pthread_attr_destroy(&attr); // Free attr
```
A common mistake

Function to be called by each thread. It will print a message that contains the thread Id.

```c
void *P_hello(void *arg){
    int* thrId = (int*)arg;
    printf("\n--> Hello from thread number: %d \n",*thrId);
    pthread_exit(NULL);
}
```
```c
int main(int argc, char *argv[]){
    /* Adapted from llnl online tutorial. A basic
    "hello world" Pthreads program showing thread
    creation + termination ------------------------*/
    #include <pthread.h>
    #include <stdio.h>
    #include <stdlib.h>
    #define NUM_THREADS 8
    pthread_t thrdNUM_THREADS[];
    int rc, t;
    for (t = 0; t < NUM_THREADS; t++) {
        rc = pthread_create(&thrdt, NULL, P_hello, &t);
        if (rc) {
            printf("ERROR: return code: %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Pay attention to argument passed to `P_hello` – last arg. of `pthread_create`
Discussion

Try Running this code.. What happens?

Try again several times. Can you explain what happens?

Also: run the drive without the last `pthread_exit`. Can you explain?
A thread can finish on its own if parent thread (e.g., main) does not need it to join at completion.

In this case: declare as ‘detached’

```c
/* --------- declare attribute */
pthread_attr_t attr;
/* --------- initialize it */
pthread_attr_init(&attr);
/* --------- set as detached */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
/* --------- create thread with attribute */
pthread_create(threadId, attr, xfun, (void *)arg);
......
/* --------- instead of join issue detach */
pthread_detach(threadId);
/* --------- do not forget: free attribute */
pthread_attr_destroy(&attr);
```

**Other functions:**

- `pthread_attr_getdetachstate` *(attr,detachstate)*
- `pthread_attr_setdetachstate` *(attr,detachstate)*
accessing shared variables requires careful control - if data is altered by a thread: If several threads modify a shared variable, we need to make sure only one thread accesses it at a time

**Mechanism: Mutual Exclusion or Mutex.**

```c
// Declare as
pthread_mutex_t mutex1;
// then set as
pthread_mutex_init(mutex1, attr);
// or with static initialization
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// lock this critical section:
pthread_mutex_lock(mutex1);
// do the work needed in this section
// Nobody else can modify variables in this section
// then unlock
pthread_mutex_unlock(mutex1)
// at completion free:
pthread_mutex_destroy(mutex1)
```
Example: parallel sum of n numbers

We want to sum the \( n \) numbers of an array \( a[0:n-1] \) by dividing the sums into \( p \) subsums which are added in a common location in memory. Shared variable SUM will contain the final sum.

... Each thread computes its subsum
... locks the code section that updates SUM
... adds subsum to SUM
... Unlocks critical section
... and exit thread.
Main program generates data (a vector) of length NUMTHRDS*VECLEN -- then generates threads that call the sum_dat function. Upon completion the result is printed. Main thread will wait for all threads to complete. This code also illustrates thread *attributes*. It sets the threads to be *joinable* (allows the main thread to join with the threads it creates).

Declarations

```
*----------------------------------------
* Illustrates the use of mutex variables
* in a threads program to sum n numbers
*----------------------------------------
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Data is passed to threads through the following struct. */
typedef struct {
    double    *a;
    double    sum;
    int        veclen;
} sumstr, *SumPtr;
sumstr SUMST;
```
```c
/* --- Define global variables and a mutex */
#define NUMTHRDS 8
#define VECLEN 50
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

/* -------------------- function sum -- activated when thread is created. */
void *sum_dat(void *arg){
  /* ------------------- local variables */
  int i, start, end, *blkNum, len ;
  double mysum, *x;
  blkNum = (int*)arg;
  len = SUMST.veclen;
  start = (*blkNum)*len;
  end = start + len;
  x = SUMST.a;

  /* ------------------- sum */
  mysum = 0;
  for (i=start; i<end ; i++)
    mysum += x[i];

  /* ------------------- Lock a mutex before updating value in the shared struct */
  pthread_mutex_lock (&mutexsum);
  SUMST.sum += mysum;

  /* ------------------- update done, unlock*/
  pthread_mutex_unlock (&mutexsum);
  printf(" -- Local sum in thread %d is %10.2f total : %10.2f\n ",
         *blkNum,mysum,SUMST.sum);
  printf(" len %d %d %d %5.2f \n",len,start,end,x[start]);

  /*------------------------ done with this thread */
  pthread_exit((void*) 0);
}
```
int main (int argc, char *argv[])
{
    /* -------------------------------main------------------------*/
    int i, n = VECLEN * NUMTHRDS, * thrNum, * status;
    double * a;
    pthread_attr_t attr;
    /* ------------------------------ alloc storage + initialize values */
    a = (double *) malloc (NUMTHRDS * VECLEN * sizeof(double));
    thrNum = (int *) malloc (NUMTHRDS * VECLEN * sizeof(double));
    for (i = 0; i < n; i++) { a[i] = (double)i;
    /* ------------------------------ have thread number in array */
    for (i = 0; i < NUMTHRDS; i++) { thrNum[i] = i;
    SUMST.veclen = VECLEN; SUMST.a = a; SUMST.sum = 0;
    /* ------------------------------ initialize mutex */
    pthread_mutex_init (&mutexsum, NULL);
    /* ------------------------------ Create threads to perform the sum */
    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. thrNum specified by 'i'; size = VECLEN */
        pthread_create (&callThd[i], &attr, sum_dat, (void*) & thrNum[i]);
    }
    pthread_attr_destroy (&attr);
    /* ------------------------------ Join to wait for the other threads */
    for (i = 0; i < NUMTHRDS; i++) {
        status = & thrNum [i];
        pthread_join (callThd[i], (void **) status);
        printf(" join number %d -- status %d \n", i, *(status));
    }
    /* ------------------------------ Now print out the sum and cleanup */
    printf("Total Sum in main thread = %.10f \n", SUMST.sum);
    free (a); free (thrNum);
    pthread_mutex_destroy (&mutexsum);
    pthread_exit (NULL);
}