# Lecture 5 :
# Pthread Programming

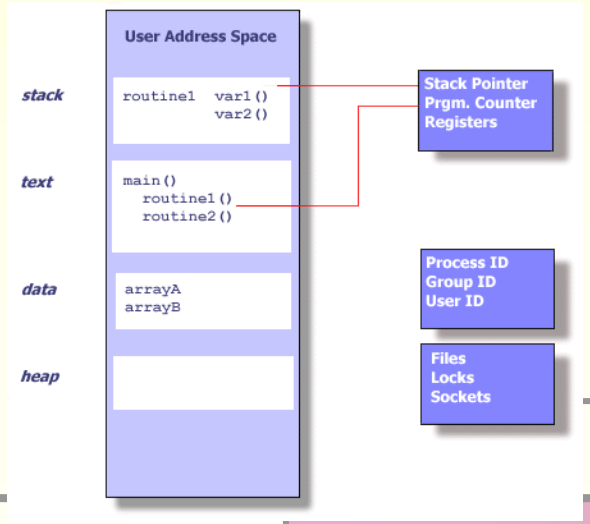
Courtesy : slides based on LLNL POSIX threads programming tutorial

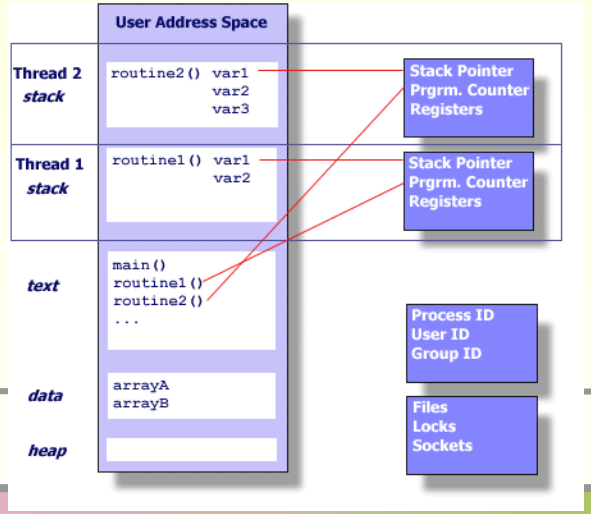
# What is Thread?

- Independent stream of instructions executed simultaneously by OS
- Multithreaded program
  - A main program contains a number of procedures that are scheduled to run simultaneously and independently by OS

# Processes and Threads

- Threads share resources of a process
  - Changes made by one thread affect other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory location is possible, and therefore require explicit synchronization by programmer
- Processes don't share resources



# Thread Properties

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies – or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- All threads within a process share same address space.
- Therefore, inter-thread communication is more efficient than inter-process communication

# pthread

- pthread
  - POSIX thread
  - Standardized C language threads for UNIX
  - For Portability
  - Working in **shared memory multiprocessor**
- Why pthreads?
  - Performance gains
  - Requires fewer system resources than process
    - Compare fork() and pthread_create() : 10~50 times

# fork() vs pthread_create()

- Performance (50,000 fork or pthread_create)

| Platform | fork() | | | pthread_create() | | |
|---|---|---|---|---|---|---|
| | real | user | sys | real | user | sys |
| Intel 2.6 GHz Xeon E5-2670 (16 cores/node) | 8.1 | 0.1 | 2.9 | 0.9 | 0.2 | 0.3 |
| Intel 2.8 GHz Xeon 5660 (12 cores/node) | 4.4 | 0.4 | 4.3 | 0.7 | 0.2 | 0.5 |
| AMD 2.3 GHz Opteron (16 cores/node) | 12.5 | 1.0 | 12.5 | 1.2 | 0.2 | 1.3 |
| AMD 2.4 GHz Opteron (8 cores/node) | 17.6 | 2.2 | 15.7 | 1.4 | 0.3 | 1.3 |
| IBM 4.0 GHz POWER6 (8 cpus/node) | 9.5 | 0.6 | 8.8 | 1.6 | 0.1 | 0.4 |
| IBM 1.9 GHz POWER5 p5-575 (8 cpus/node) | 64.2 | 30.7 | 27.6 | 1.7 | 0.6 | 1.1 |
| IBM 1.5 GHz POWER4 (8 cpus/node) | 104.5 | 48.6 | 47.2 | 2.1 | 1.0 | 1.5 |
| INTEL 2.4 GHz Xeon (2 cpus/node) | 54.9 | 1.5 | 20.8 | 1.6 | 0.7 | 0.9 |
| INTEL 1.4 GHz Itanium2 (4 cpus/node) | 54.5 | 1.1 | 22.2 | 2.0 | 1.2 | 0.6 |

# Pthreads API

- Three groups
  - Thread Management
    - Thread creation, and destruction
  - Mutexes (mutual exclusion)
    - synchronization
  - Conditional Variables
    - Communication between threads that share a mutex

# Thread Management

- `pthread_create (thread,attr,start_routine,arg)`
- `pthread_exit (status)`
- `pthread_cancel (thread)`

# Thread Creation

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

- Creates a new thread and makes it executable.
- The creating process (or thread) must provide a location for storage of the thread id.
- The third parameter is just the name of the function for the thread to run.
- The last parameter is a pointer to the arguments.

# Thread Creation

- When a new thread is created, it runs concurrently with the creating process.
- When creating a thread, you indicate which function the thread should execute.
- Thread handle returned via **`pthread_t`** structure
- Specify **`NULL`** to use default attributes
- Single argument sent to the function
- If no arguments to function, specify **`NULL`**
- Check error codes!

# Thread Termination

- There are several ways in which a pthread may be terminated:
  - The thread returns from its starting routine (the main routine for the initial thread).
  - The thread makes a call to the **`pthread_exit`** subroutine.
  - The thread is canceled by another thread via the **`pthread_cancel`** routine
  - The entire process is terminated due to making a call to either the exec() or exit()
  - If main() finishes first, without calling **`pthread_exit`** explicitly itself

# Thread Termination

```
void pthread_exit(void *value_ptr);
```

- Typically, the pthread_exit() routine is called to quit the thread.
- If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.
- The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.
- Cleanup: the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

# Thread Cancellation

- One thread can request that another exit with pthread_cancel
- **`int pthread_cancel(pthread_t thread);`**
- The pthread_cancel returns after making the request.

# Example 1: Thread Creation

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

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

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

# Example 2: Passing Parameters to Thread

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

void *PrintHello(void *ptr)
{
   char *filename;
   int j;
   filename = (char *) ptr;
   while (1) {
      printf("Hello World! It's me, thread %s!\n", filename);
      sleep(1);
   }
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t thread[100];
   int err_code, i=0;
   char *filename;

   printf ("Enter thread name at any time to create thread\n");

   while (1) {
      filename = (char *) malloc (80*sizeof(char));
      scanf ("%s", filename);
      printf("In main: creating thread %d\n", i);
      err_code = pthread_create(&thread[i], NULL, PrintHello, (void *)filename);
      if (err_code){
         printf("ERROR code is %d\n", err_code);
         exit(-1);
      } else i++;
   }
   pthread_exit(NULL);
}
```

# Example 3: Files

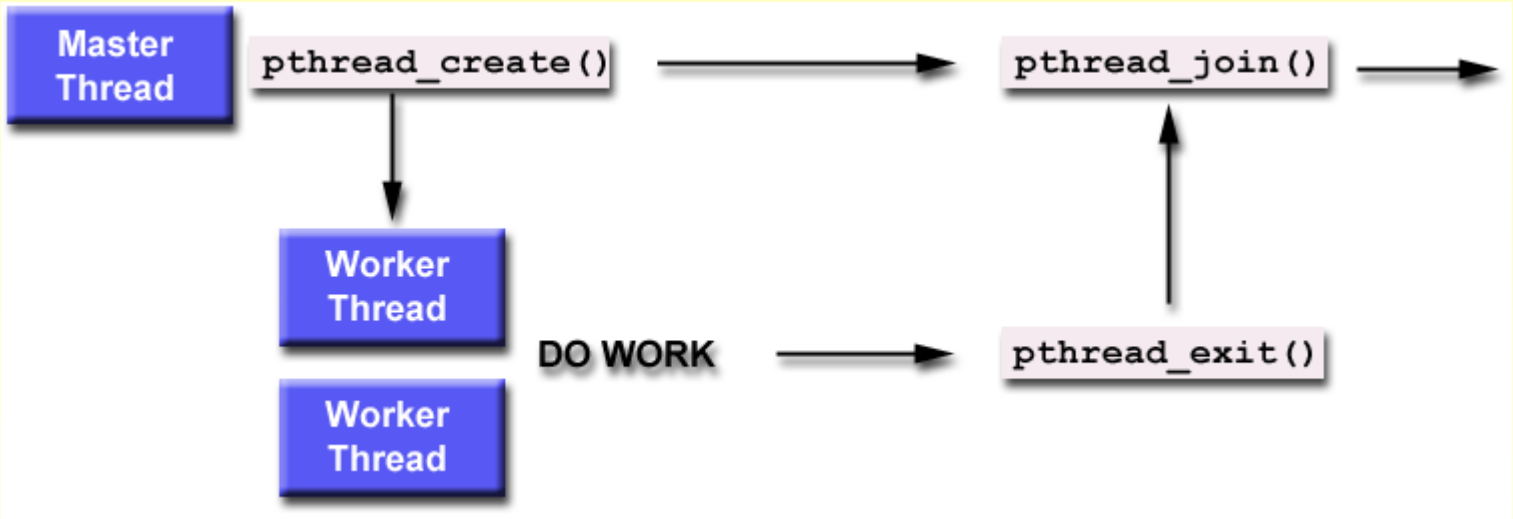
```
void *PrintHello(void *ptr)
{
   FILE *file;
   char *filename;
   char textline[100];
   int j;

   filename = (char *) ptr;
   printf("Hello World! Opening %s!\n", filename);
   file = fopen(filename, "r");
   if (file != NULL) {
     while(fscanf(file, "%s\n", textline) != EOF) printf ("%s\n", textline);
   }
   fclose(file);
}
```

# JOINING

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- The pthread_join() subroutine blocks the calling thread until the specified thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
- A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

# JOINING


Master Thread
pthread_create()
pthread_join()
Worker Thread
DO WORK
pthread_exit()
Worker Thread

# Example 4: JOIN

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}
```

```
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   int rc;
   long t;
   void *status;

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
         }
      }

   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
   }
   printf("Main: program completed. Exiting.\n");
   pthread_exit(NULL);
}
```

# Mutexes

- Mutual Exclusion
- implementing thread synchronization and protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource
  - only one thread can lock (or own) a mutex variable at any given time
- Used for preventing race condition
- When several threads compete for a mutex, the losers block at that call – an unblocking call is available with "trylock" instead of the "lock" call.

# Mutex Routines

- **`pthread_mutex_init (mutex,attr)`**
- **`pthread_mutex_destroy (mutex)`**

- Mutex variables must be declared with type **`pthread_mutex_t`**, and must be initialized before they can be used

# Locking/Unlocking Mutexes

- **`pthread_mutex_lock (mutex)`**
  - acquire a lock on the specified *mutex* variable
- **`pthread_mutex_trylock (mutex)`**
  - attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code
- **`pthread_mutex_unlock (mutex)`**
  - unlock a mutex if called by the owning thread

# User's Responsibility for Using Mutex

- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so.

- For example, if 3 threads are updating the same data, but only one or two use a mutex, the data can still be corrupted.

| Thread 1 | Thread 2 | Thread 3 |
|---|---|---|
| Lock | Lock | |
| A = 2 | A = A+1 | A = A*B |
| Unlock | Unlock | |

# Example 5: Mutexes

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

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

#define NUMTHRDS 4
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

void *dotprod(void *arg)
{
   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;
   mysum = 0;
   for (i=start; i<end ; i++) mysum += (x[i] * y[i]);
   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);
}
```

# Example 5: Mutexes

```
int main (int argc, char *argv[])
{
  long i;
  double *a, *b;
  void *status;

  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++) pthread_create(&callThd[i], &attr, dotprod, (void *)i);

  for(i=0;i<NUMTHRDS;i++) pthread_join(callThd[i], &status);
  printf ("Sum =  %f \n", dotstr.sum);

  free (a);
  free (b);
  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```

# Condition Variables

- another way for threads to synchronize
- mutexes
  - synchronization by controlling thread access to data
- condition variables
  - synchronization based upon the actual value of data.
  - Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met
  - always used in conjunction with a mutex lock

# Condition Variables Routines

- **`pthread_cond_init (condition,attr)`**
- **`pthread_cond_destroy (condition)`**

- Condition variables must be declared with type **`pthread_cond_t`**, and must be initialized before they can be used.

- *attr* is used to set condition variable attributes.(NULL: defaults)

- **`pthread_cond_destroy()`** should be used to free a condition variable that is no longer needed.

# Condition Variables Routines

- **`pthread_cond_wait (condition,mutex)`**
  - blocks the calling thread until the specified *condition* is signalled.
  - This routine should be called while *mutex* is locked
  - will automatically release the mutex lock while it waits
  - After signal is received and thread is awakened, *mutex* will be automatically locked for use
- **`pthread_cond_signal (condition)`**
  - signal (or wake up) another thread which is waiting on the condition variable.
  - It is a logical error to call pthread_cond_signal() before calling pthread_cond_wait().
- **`pthread_cond_broadcast (condition)`**
  - should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

# Example 6: Control Variables

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

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

int     count = 0;
int     thread_ids[3] = {0,1,2};
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++;

    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      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);

    sleep(1);
  }
  pthread_exit(NULL);
}
```

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

  printf("Starting watch_count(): thread %ld\n", my_id);
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
  }
  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];
  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);

  for (i=0; i<NUM_THREADS; i++) pthread_join(threads[i], NULL);
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```