Introduction to Pthreads

A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task.
In the traditional UNIX model, when a process needs something performed by another entity, it forks a child process and lets the child perform the processing. Most network servers under Unix, for example, are written this way. Although this paradigm has served well for many years, there are problems with fork:

- fork is expensive. Memory is copied from the parent to the child, all descriptors are duplicated in the child, and so on.
- Interprocess communication (IPC) is required to pass information between the parent and child after the fork. Information from the parent to the child before the fork is easy, since the child starts with a copy of the parent's data space and with a copy of all the parent's descriptors. But returning information from the child to the parent takes more work.

Threads help with both problems. Threads are sometimes called lightweight processes, since a thread is "lighter weight" than a process. That is, thread creation can be 10-100 times faster than process creation.

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes the problem of ***synchronization.*** But more than just the global variables are shared.

Benefits of threads:

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type. Each thread can then handle its event using a synchronous programming model. A synchronous programming model is much simpler than an asynchronous one.
- Some problems can be partitioned so that overall program throughput can be improved. A single process that has multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control. With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on the processing performed by each other.
- Similarly, interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

All threads within a process share:

- process instructions,
- most data,
- open files (e.g., descriptors),
- signal handlers and signal dispositions,
- current working directory, and
- user and group IDS.

But each thread has its own:

- thread ID,
- set of registers, including program counter and stack pointer,
- stack (for local variables and return addresses),
- errno,
- signal mask, and
- priority.

pthread_create Function:

When a program is started by exec, a single thread is created, called the ***initial thread*** or ***main thread.*** The pthread_create() function creates a new thread. The new thread commences execution by calling the function identified by start with the argument arg (i.e., start(arg)). The thread that calls pthread_create() continues execution with the next statement that follows the call.
The **arg argument** is declared as void *, meaning that we can pass a pointer to any type of object to the start function. Typically, arg points to a global or heap variable, but it can also be specified as NULL. If we need to pass multiple arguments to start, then arg can be specified as a pointer to a structure containing the arguments as separate fields. With judicious casting, we can even specify arg as an int.

```
#include<stdio.h>
main()
{
int myval = 5;
void* ptr = (void*)myval;
printf("%d",(int)ptr);
}
/*[root@ParallelProcessingLab klupplab]# ./a.out
5 */
```

The **return value of start** is likewise of type void *, and it can be employed in the same way as the arg argument.
The **thread argument** points to a buffer of type pthread_t into which the unique identifier for this thread is copied before pthread_create() returns. This identifier can be used in later Pthreads calls to refer to the thread.

Each thread has numerous ***attributes:*** its priority, its initial stack size, whether it should be a daemon thread or not, and so on. When a thread is created, we can specify these attributes by initializing a pthread_attr_t variable that overrides the default. We normally take the default, in which case, we specify the ***attr*** argument as a null pointer.

The **return value** from the Pthread functions is normally 0 if OK or nonzero on an error.

Pthread_self Function:

Each thread has an ID that identifies it within a given process. The thread ID is returned by **pthread_create,** and we saw that it was used by **pthread_join.** A thread fetches this value for itself using **pthread_self.**

Pthread_j oin Function:

We can wait for a given thread to terminate by calling **pthread_join.** Comparing threads to Unix processes, **pthread_create** is similar to **fork,** and **pthread_join** is similar to **waitpid.**

We must specify the tid of the thread for which we wish to wait. Unfortunately, we have no way to wait for any of our threads (similar to **waitpid** with a process ID argument of -1).

If the status pointer is nonnull, the return value from the thread (a pointer to some object) is stored in the location pointed to by status.

Pthread_detach Function:

A thread is either ***joinable*** (the default) or ***detached.*** When a joinable thread terminates, its thread ID and exit status are retained until another thread in the process calls pthread_join. But a detached thread is l i e a daemon process: when it terminates, all its resources are released, and we cannot wait for it to terminate. If one thread needs to know when another thread terminates, it is best to leave the thread as joinable. The pthread_detach function changes the specified thread so that it is detached. This function is commonly called by the thread that wants to detach itself, as in **Pthread_detach(pthread_self());**

pthread_exit Function:

One way for a thread to terminate is to call pthread_exit.

If the thread is not detached, its thread ID and exit status are retained for a later pthread_join by some other thread in the calling process. The pointer ***status*** must not point to an object that is local to the calling thread (e.g., an automatic variable in the thread start function), since that object disappears when **the** thread terminates.

A thread can terminate in two other ways:

- The function that started the thread (the third argument to pthread_create) can return. Since this function must be declared as returning a void pointer, that return value is the exit status of the thread.
- If the main function of the process returns or if any thread calls exit ( ) or _exit ( ), the process terminates immediately, including any threads that are still running.

```
#include <pthread.h>
void* worker( void* p ) {
  int* ip = (int*)p;
  printf("Hello world from worker
%i!\n",*ip);
}
int main() {
  pthread_t OtherThread[4];
  int i;
  for(i=0;i<4;i++) {
    pthread_create( &OtherThread[i],
NULL, worker, &i );
    sleep(1);
  }
}
/*
[vishnu@mannava PP]$ ./a.out
Hello world from worker 0!
Hello world from worker 1!
Hello world from worker 2!
Hello world from worker 3!
*/
```

```
#include <pthread.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
 printf("\n%d: Hello World!\n", threadid);
 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("Creating thread %d\n", t);
         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
         if (rc) {
            printf("ERROR; pthread_create() return code is %d\n", rc);
            exit(-1);
                  }
    }
     for(t=0;t < NUM_THREADS;t++)
     {
     pthread_join( threads[t], NULL);
     printf("Joining thread %d\n", t);
     }
    return 0;
}/*[vishnu@mannava PP]$ ./a.out
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
Creating thread 4
2: Hello World!
3: Hello World!
4: Hello World!
Joining thread 0
Joining thread 1
Joining thread 2
Joining thread 3
Joining thread 4*/
```

<table>
<tr><th colspan="3">Comparison of process and thread primitives</th></tr>
<tr><th>Process primitive</th><th>Thread primitive</th><th>Description</th></tr>
<tr><td>fork</td><td>pthread_create</td><td>create a new flow of control</td></tr>
<tr><td colspan="3">#include <unistd.h>
pid_t fork(void);
Returns: 0 in child, process ID of child in parent, -1 on error
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);
Returns 0 on success, or a positive error number on error</td></tr>
<tr><td>exit</td><td>pthread_exit</td><td>exit from an existing flow of control</td></tr>
<tr><td colspan="3">#include <stdlib.h>
void exit(int status);
#include <pthread.h>
void pthread_exit(void *rval_ptr);</td></tr>
<tr><td>waitpid</td><td>pthread_join</td><td>get exit status from flow of control</td></tr>
<tr><td colspan="3">#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
Both return: process ID if OK, 0 (see later), or -1 on error
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);</td></tr>
<tr><td>getpid</td><td>pthread_self</td><td>get ID for flow of control</td></tr>
<tr><td colspan="3">pid_t getpid(void);
Returns: process ID of calling process
pid_t getppid(void);
Returns: parent process ID of calling process
#include <pthread.h>
pthread_t pthread_self(void);
Returns: the thread ID of the calling thread</td></tr>
<tr><td>abort</td><td>pthread_cancel</td><td>request abnormal termination of flow of control</td></tr>
<tr><td colspan="3">#void abort(void);
include <stdlib.h>
#include <pthread.h>
int pthread_cancel(pthread_t tid);</td></tr>
</table>

```
int pthread_detach(pthread_t tid);
int pthread_equal(pthread_t tid1, pthread_t tid2);
Returns: nonzero if equal, 0 otherwise
```

Thread Synchronization

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Pthreads data types	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_attr_t	Pthread attributes object

```
/* File:  pth_hello.c
Purpose: Illustrate basic use of pthreads:  create some threads, each of which prints a
message. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
const int MAX_THREADS = 64;
/* Global variable:  accessible to all threads */
int thread_count;
void Usage(char* prog_name);
void *Hello(void* rank);  /* Thread function */
/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
   /* Get number of threads from command line */
   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
   thread_handles = malloc (thread_count*sizeof(pthread_t));
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
   printf("Hello from the main thread\n");
   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   free(thread_handles);
   return 0;
}  /* main */
/*-------------------------------------------------------------------*/
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */
   printf("Hello from thread %ld of %d\n", my_rank, thread_count);
   return NULL;
}  /* Hello */
/*-------------------------------------------------------------------*/
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "0 < number of threads <= %d\n", MAX_THREADS);
   exit(0);
}  /* Usage */

/*[vishnu@mannava PP]$ cc -o pth_hello pth_hello.c -lpthread
[vishnu@mannava PP]$./pth_hello 4
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4 */
```

/*pthreads complex Passing arguments demo program */

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   8
char *messages[NUM_THREADS];
typedef struct thread_data {
   int thread_id;
   int sum;
   char *message;
} tdata_t;
void *PrintHello(void *threadarg) {
   int taskid, sum;
   char *hello_msg;
   struct tdata_t *my_data;
   //sleep(1);
   my_data = (tdata_t *) 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);
   free(threadarg);
   pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t threads[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++) {
       tdata_t *tdata = (tdata_t *) malloc(sizeof(tdata_t));
      sum = sum + t;
      tdata->thread_id = t;
      tdata->sum = sum;
      tdata->message = messages[t];
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) tdata);
      if (rc) {
          printf("ERR; pthread_create() ret = %d\n", rc);
          exit(-1);
      }
  }
  return 0;
}
/*
[vishnu@mannava PP]$ ./a.out
Creating thread 0
Creating thread 1
Task 0: English: Hello World!
Creating thread 2
Creating thread 3
Task 2: Spanish: Hola al mundo
Task 1: French: Bonjour, le monde!
Creating thread 4
Task 3: Klingon: Nuq neH!
Creating thread 5
Creating thread 6
Creating thread 7
Task 6: Japan: Sekai e konnichiwa!
Task 4: German: Guten Tag, Welt!
/*  A simple child/parent signaling example. - main-signal.c      */
#include <stdio.h>
#include <pthread.h>
int done = 0;
void* worker(void* arg) {
    printf("this should print first\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create(&p, NULL, worker, NULL);
    while (done == 0)
       ;
    printf("this should print last\n");
    return 0;
}
/*
```

```
vishnu@mannava:~/threads$ cc main-signal.c -lpthread
vishnu@mannava:~/threads$ ./a.out
this should print first
this should print last
*/

/* A more efficient signaling via condition variables. - main-signal-cv.c    */
#include <stdio.h>
#include <pthread.h>
/* simple synchronizer: allows one thread to wait for another structure
"synchronizer_t" has all the needed data methods are:
   init (called by one thread)
   wait (to wait for a thread)
   done (to indicate thread is done)      */
typedef struct __synchronizer_t {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int done;
} synchronizer_t;

synchronizer_t s;

void signal_init(synchronizer_t *s) {
    pthread_mutex_init(&s->lock, NULL);
    pthread_cond_init(&s->cond, NULL);
    s->done = 0;
}

void signal_done(synchronizer_t *s) {
    pthread_mutex_lock(&s->lock);
    s->done = 1;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}

void signal_wait(synchronizer_t *s) {
    pthread_mutex_lock(&s->lock);
    while (s->done == 0)
      pthread_cond_wait(&s->cond, &s->lock);
    pthread_mutex_unlock(&s->lock);
}

void* worker(void* arg) {
    printf("this should print first\n");
    signal_done(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    signal_init(&s);
    pthread_create(&p, NULL, worker, NULL);
    signal_wait(&s);
    printf("this should print last\n");

    return 0;
}
/*
vishnu@mannava:~/threads$ cc main-signal-cv.c -lpthread
vishnu@mannava:~/threads$ ./a.out
this should print first
this should print last
*/
```