Posix Threads

Thread state

- Each thread has its own stack and local variables
- Globals are shared.
- File descriptors are shared. If one thread closes a file, all other threads can’t use
- The file I/O operations block the calling thread.
- Some other functions act on the whole process.
  - For example, the exit() function operates terminates the entire and all associated threads.

Thread Implementation

- The most important of these APIs, in the Unix world, is the one developed by the group known as POSIX.
- POSIX is a standard API supported
- Portable across most UNIX platforms.
- PTHREAD library contains implementation of POSIX standard
- To link this library to your program use ***–lpthread***
  - **`gcc MyThreads.c -o MyThreadExecutable - lpthread`**

Thread Creation

- **`pthread_create( pthread_t *threadid ,const pthread_attr_t *attr, void *(*start_routine)(void *),void *arg);`**
  - This routine creates a new thread and makes it executable.
  - Thread stack is allocated and thread control block is created
  - Once created, a thread may create other threads.
  - Note that an "initial thread" exists by default and is the thread which runs main.
  - Returns zero, if ok
  - Returns Non-zero if error
  - **threadid**
    - The routine returns the new thread ID via the **threadid**
    - The caller can use this thread ID to perform various operations
    - This ID should be checked to ensure that the thread was successfully created.

Thread Creation

- `pthread_create( pthread_t *threadid ,const pthread_attr_t *attr, void *(*start_routine)(void *),void *arg);`
- **attr**
  - used to set thread attributes.
  - NULL for the default values.
- **start_routine**
  - The C routine that the thread will execute once it is created.
- **arg**
  - Arguments are passed to *start_routine* via *arg*.
  - Arguments must be passed by reference as pointers
  - These pointers must be cast as pointers of type void.

Attribute Object

- When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:
  - It is non-detached
  - It has a a default stack and stack size
  - It inherits the parent's priority

Thread Termination

- **`pthread_exit(status)`**
- Several ways of termination:
  - 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 a call to the exit subroutines.

Thread Termination

- The **`pthread_exit()`** routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- If the "initial thread" exits with **`pthread_exit()`** instead of exit(), other threads will continue to execute.
- The programmer may specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread i.e wait for this thread.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void * PrintHello(void *threadid)
{   printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}
int main()
{
   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; return code from pthread_create() is %d\n", rc);
       exit(-1);
    }
 }
pthread_exit(NULL); }
```

Passing Arguments To Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- What if we want to pass multiple arguments.
- Create a structure which contains all of the arguments
- Pass a pointer to the structure in the pthread_create() routine.
- Argument must be passed by reference and cast to (void *).
- Important:
  - Threads initially access their data structures in the parent thread's memory space.
  - That data structure must not be corrupted/modified until the thread has finished accessing it.

Incorrect pthread_create() argument passing

```
for(t=0;t < NUM_THREADS;t++)
{
 printf("Creating thread %d\n", t);
 rc = pthread_create(&threads[t],
 NULL, printHello, (void *) &t);
 ...
 }
```

Correct pthread_create() argument passing

```
int *task_ids[NUM_THREADS];
for(t=0;t < NUM_THREADS;t++)
{
  task_ids[t] = new int;
  *task_ids[t] = t;
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL,
  PrintHello, (void *) task_ids[t]);
  ...
}
```

Passing a structure as an argument

```
struct thread_data
{ int thread_id;
  int sum;
};
thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg)
{ thread_data *my_data;
  ...
  my_data = (struct thread_data *) threadarg;
  taskid = my_data->thread_id;
  sum = my_data->sum;
  ...
}
```

Passing a structure as an argument

```
int main()
{ ...
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  rc = pthread_create(&threads[t], NULL,
  PrintHello, (void )&thread_data_array[t])
  ;
  ...
}
```

Thread ID

- **`pthread_self()`**
  - Returns the unique thread ID of the calling thread.
- **`pthread_equal(threadid1,threadid2)`**
  - Compares two thread IDs:
    - If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Thread Suspension and Termination

- Similar to UNIX processes, threads have the equivalent of the wait() and exit() system calls
- pthread_join()
  - Used to block threads
- pthread_exit()
  - Used to terminate threads
- To instruct a thread to block and wait for a thread to complete, use the **pthread_join()** function.
- This is also the mechanism used to get a return value from a thread
- Any thread can call join on (and hence wait for) any other thread.

Detached State

- Each thread can be either *joinable* or *detached*.
- **Detached:** on termination all thread resources are released by the OS.
- A detached thread cannot be joined.
- No way to get at the return value of the thread.

Joining thread

- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- Joining a thread means waiting for a thread
- **pthread_join(threadid, status)**
  - "Joining" is one way to accomplish synchronization between threads.
  - subroutine blocks the calling thread until the specified *threadid* thread terminates.
    - The programmer is able to obtain the target thread's termination return status (if specified) in the *status* parameter.
- It is impossible to join a detached thread

Joining thread

- Multiple threads cannot wait for the same thread to terminate.
- If they try to, one thread returns successfully
- The others fail with an error of ESRCH
- After pthread_join() returns, any stack storage associated with the thread can be reclaimed by the application.
- Threads which have exited but have not been joined are equivalent to zombie processes.
- Their resources cannot be fully recovered.

pthread_detach()

- By default threads are created joinable.
- Instead of waiting for a child, a parent thread can specify that
  - it does not require a return value
  - or any explicit synchronization with that thread.
- To do this, the parent thread uses the **pthread_detach()** function.
- After this call, there is no thread waiting for the child – it executes independently until termination.
- To avoid memory leaks a thread should
  - either be *joined*
  - Or detached by a call to **pthread_detach()**
- int pthread_detach(pthread_t tid);
  - Returns 0 on OK, nonzero on error
- Threads can detach themselves by calling *pthread_detach* with an argument of *pthread_self*