

Pthread

Prof. Joonwon Lee(joonwon@skku.edu)
TA – Taekyun Roh(throh0198@gmail.com)
Sewan Ha(hsewan@gmail.com)
Sungkyunkwan University
<http://csl.skku.edu>



The Pthread API

- **ANSI/IEEE POSIX1003.1-1995 Standard**
- **Thread management**
 - Work directly on threads – creating, terminating, joining, etc.
 - Include functions to set/query thread attributes.
- **Mutexes(Mutual Exclusion)**
 - Provide for creating, destroying, locking and unlocking mutexes.
- **Conditional variables**
 - Include functions to create, destroy, wait and signal based upon specified variable values.

What is thread?



Multi tasking

Process

thread

thread

thread

Process

thread

thread

thread

Process

thread

thread

thread

Process

thread

thread

thread

Creating Threads (1)

- **int pthread_create (pthread_t *thread_id, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)**
 - **pthread_create()** returns the new thread ID via the **thread** argument.
 - The caller can use this thread ID to perform various operations on the thread.
 - The **attr** parameter is used to set thread attributes.
 - NULL for the default values.
 - The **start_routine** denotes the C routine that the thread will execute once it is created.
 - C routine that the thread will execute once it is created.
 - A single argument may be passed to **start_routine()** via **arg**.

Creating Threads (2)

■ Notes:

- Initially, **main()** comprises a single, default thread.
- All other threads should must be explicitly created by the programmer.
- Once created, threads are peers, and may create other threads.
- The maximum number of threads that may be created by a process is implementation dependent.

Terminating Threads

- **void pthread_exit (void *retval)**
 - **pthread_exit()** terminates the execution of the calling thread.
 - Typically, this is called after a thread has completed its work and is no longer required to exist.
 - The **retval** argument is the return value of the thread.
 - It can be consulted from another thread using **pthread_join()**.
 - It does not close files; any files opened inside the thread will remain open after the thread is terminated.

Cancelling Threads

- **int pthread_cancel (pthread_t thread)**
 - **pthread_cancel()** sends a cancellation request to the thread denoted by the **thread** argument.
 - Depending on its settings, the target thread can then either ignore request, honor it immediately, or defer it till it reaches a cancellation point.
 - pthread_setcancelstate():
PTHREAD_CANCEL_(ENABLE|DISABLE)
 - pthread_setcanceltype():
PTHREAD_CANCEL_(DEFERRED|ASYNCHRONOUS)
 - Threads are always created by **pthread_create()** with cancellation enabled and deferred.

Joining Threads

- **int pthread_join (pthread_t thread, void **retval)**
 - **pthread_join()** suspends the execution of the calling thread until the thread identified by **thread** terminates, either by calling **pthread_exit()** or by being cancelled.
 - The return value of **thread** is stored in the location pointed by **retval**.
 - It returns **PTHREAD_CANCELLED** if thread was cancelled.
 - It is impossible to join a detached thread.

Detaching Threads

- **int pthread_detach (pthread_t thread)**
 - **pthread_detach()** puts the thread in the detached state.
 - This guarantees that the memory resources consumed by **thread** will be freed immediately when thread terminates.
 - However, this prevents other threads from synchronizing on the termination of thread using **pthread_join()**.
 - A thread can be detached when it is created:

```
pthread_t tid;  
pthread_attr_t attr;  
  
pthread_attr_init (&attr);  
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);  
pthread_create(&tid, &attr, start_routine, NULL);  
pthread_attr_destroy (&attr);
```

Thread Identifiers

- **pthread_t pthread_self (void)**
 - **pthread_self()** returns the unique, system assigned thread ID of the calling thread.
- **int pthread_equal (pthread_t t1, pthread_t t2)**
 - **pthread_equal()** returns a non-zero value if **t1** and **t2** refer to the same thread.
 - Because thread IDs are opaque objects, the C language equivalence operator **==** should not be used to compare two thread IDs against each other.

Thread API List



▪ Thread Synchronization Function

- Pthread_mutex_init
- Pthread_mutex_destroty
- Pthread_mutex_lock
- Pthread_mutex_unlock
- Pthread_cond_init
- Pthread_cond_signal
- Pthread_cond_broadcast
- Pthread_cond_wait
- Pthread_cond_timewait
- Pthread_cond_destroy

Thread API List



▪ Thread Attribute Function

- Pthread_attr_init
- Pthread_attr_destroy
- Pthread_attr_getscope
- Pthread_attr_setscope
- Pthread_attr_getdetachstate
- Pthread_attr_setdetachstate

Thread API List



▪ Thread Signal Function

- Pthread_sigmask
- Pthread_kill
- sigwait

▪ Thread Cancel Function

- Pthread_cancel
- Pthread_setcancelstate
- Pthread_setcanceltype
- Pthread_testcancel

Exercise 1

- **Make a program to sum**

- Input one number when you execute the program and calculate the summation until your input number.

```
int sum;
void *sum(void *value){
    int i, upper = strtol(value, 0, 0); // string to long

    //calculate the number and exit the thread

}
Int main(int argc, char *argv[]){
    pthread_t tid;

    if(argc < 2){
        printf("Usage: %s number\n", argv[0]);
        exit(-1);
    }
    //thread create & join (if you want to detach, you can use.)

    printf("sum = %d\n",sum);
}
```

gcc thread.c -o thread -lpthread

Exercise 2

■ Make a program

- Print own information three times such as
[thread_name] pid:123 tid: 213a23

```
void *thread_function(void *t_name)
{
    pid_t pid;           // process id
    pthread_t tid;       // thread id
    pid = getpid();
    tid = pthread_self();
    char* thread_name = (char*)t_name;
    int i = 0;

    while (i<3) {
        //print the information of thread name, pid, tid
    }
}
```

gcc thread.c -o thread -lpthread

Exercise 2

```
int main(){
    pthread_t p_thread[2];
    int thr_id;
    int status;
    char p_1[] = "thread_1"; // first thread name
    char p_2[] = "thread_2"; // second thread name
    char p_M[] = "thread_m"; // main thread name

    sleep(1);
    //create first thread

    if (thr_id < 0){
        // If thread create successfulluy using pthread_create(), return 0
    }
    //Create second thread
    thr_id = pthread_create(&p_thread[1], NULL, thread_function, (void *)p_2);
    if (thr_id < 0){
        // If thread create successfulluy using pthread_create(), return 0
    }

    //main() function also execute function

    // wait for child process thread
}
```