MULTI-THREAD SERVER

pthread_create()

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*routine) (void *), void *arg);
```

- Create a new thread
- Parameters:
  - [OUT] `tid`: points to ID of the new thread
  - [IN] `attr`: points to structure whose contents are used to determine attributes for the new thread
  - [IN] `routine`: the new thread starts execution by invoking `routine()`
  - [IN] `arg`: points to the argument is passed as the sole argument of `routine()`
- Return:
  - On success, returns 0
  - On error, returns an error number
- Compile and link with *-pthread*

`pthread_create()`

- By default, the new thread is joinable:
  - Not automatically cleaned up by GNU/Linux when it terminates
  - the thread's exit state hangs around in the system until another thread calls `pthread_join()` to obtain its return value
- Detached thread is cleaned up automatically when it terminates
  - Another thread may not obtain its return value
- Detach a thread: `int pthread_detach(pthread_t tid)`
  - On success, returns 0
  - On error, returns an error number

Multi-thread TCP Echo Server

```
pthread_t tid;
int listenfd, connfd;
//Step 1: Construct socket
//Step 2: Bind address to socket
//Step 3: Listen request from client

//Step 4: Communicate with client
while (1) {
   connfd = accept (listenfd, ... );
   pthread_create(&tid, NULL, &client_handler,
                        (void *) connfd);
}
close(listenfd);
return 0;
```

Multi-thread TCP Echo Server(cont.)

```
void *client_handler(void *arg){
    int clientfd;
    int sendBytes, rcvBytes;
    char buff[BUFF_SIZE];

    pthread_detach(pthread_self());
    clientfd = (int) arg;
    while(1){
        rcvBytes = recv(clientfd, buff, BUFF_SIZE, 0);
        if (rcvBytes < 0){
            perror("\nError: ");
            break;
        }
        sendBytes = send(clientfd, buff, rcvBytes,0);
        if (sendBytes < 0){
            printf("\nError:");
            break;
        }
    }
    close(clientfd);
}
```

Synchronize threads

- Since multiple threads can be running concurrently, accessing the shared variables:
  - The order of the accessing shared memory is unpredictable, so
  - The processing flow of the thread may be incontrollable, and/or
  - The process crash
- Synchronize threads so that only one thread can access shared meory:
  - Inter-lock
  - Semaphore
  - Mutex

Mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mptr);
int pthread_mutex_unlock(pthread_mutex_t * mptr);
```

- The thread can access the shared variable only when it hold the mutex
- `pthread_mutex_lock()`: lock a mutex
- `pthread_mutex_unlock()`: unlock a mutex
- If the thread try to lock a mutex that is already locked by some other thread, it is blocked until the mutex is unlocked.

```
void *routine(void *arg){
    //...
    pthread_mutex_lock(mptr);
    // access shared memory
    pthread_mutex_unlock(mptr);
//...
}
```

`fork()` **VS** `pthread_create()`

`fork()`

- Heavy-weight
- Passing information from the parent to the child before the fork is easy
- Returning information from the child to the parent takes more work
- Needn’t synchronize processes
- Greater isolation between the parent and the child

`pthread_create()`

- Light-weight
- Passing information from a thread to the others is easy
- Don’t need signal-driven processing when the threads ends.
- May synchronize threads
- If a thread crashes, process may crash