# Operating Systems

# Project #5

2019041094 김정민

본인이 설계한 스레드풀 알고리즘 (주석 참조)

```c
static void *worker(void *param)
{
    pthread_pool_t *pool = (pthread_pool_t *)param;

    while (1) {
        pthread_mutex_lock(&(pool->mutex));

        // 대기열에 작업이 없는 경우
        while (pool->q_len == 0 && pool->running) {
            pthread_cond_wait(&(pool->empty), &(pool->mutex));
        }

        // 스레드풀 종료 조건 확인
        if (!(pool->running) || (pool->q_len) == 0) {
            pthread_mutex_unlock(&(pool->mutex));
            pthread_exit(NULL);
            break;
        }

        // 작업을 꺼내서 실행
        task_t task = pool->q[pool->q_front];
        pool->q_front = (pool->q_front + 1) % pool->q_size;
        pool->q_len--;

        // 대기열에 작업을 사용했으니 full 조건 변수를 깨워준다.
        pthread_cond_signal(&(pool->full));

        pthread_mutex_unlock(&(pool->mutex));

        (task.function)(task.param);
    }

    (pool->bee_size)--;

    pthread_mutex_unlock(&(pool->mutex));

    // 스레드를 종료합니다.
    pthread_exit(NULL);

    return NULL;
}
```

Worker 함수

```c
int pthread_pool_init(pthread_pool_t *pool, size_t bee_size, size_t queue_size)
{
    if (bee_size > POOL_MAXBSIZE || queue_size > POOL_MAXQSIZE) {
        return POOL_FAIL;
    }

    // 대기열로 사용할 원형 버퍼의 크기가 일꾼 스레드의 수보다 작으면 효율을 극대화할 수 없다.
    // 사용자가 요청한 queue_size가 최소한 bee_size 되도록 자동으로 상향 조정한다.
    if (bee_size > queue_size) queue_size = bee_size;

    // 스레드풀 정보 초기화
    pool->running = true;
    pool->q = (task_t *)malloc(queue_size * sizeof(task_t));
    pool->q_size = queue_size;
    pool->q_front = 0;
    pool->q_len = 0;
    pool->bee = (pthread_t *)malloc(bee_size * sizeof(pthread_t));
    pool->bee_size = bee_size;

    // 락과 조건변수 초기화
    pthread_mutex_init(&(pool->mutex), NULL);
    pthread_cond_init(&(pool->full), NULL);
    pthread_cond_init(&(pool->empty), NULL);

    // 일꾼 스레드 생성
    for (int i = 0; i < bee_size; i++) {
        if (pthread_create(&(pool->bee[i]), NULL, worker, pool) != 0) {
            pthread_pool_shutdown(pool, POOL_DISCARD);
            return POOL_FAIL;
        }

    }

    return POOL_SUCCESS;
}
```

Pthread_pool_init 함수

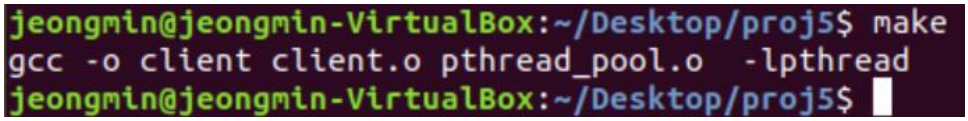
```c
int pthread_pool_submit(pthread_pool_t *pool, void (*f)(void *p), void *p, int flag)
{
    pthread_mutex_lock(&(pool->mutex));

    // 대기열이 꽉 찬 경우
    if (pool->q_len == pool->q_size) {
        if (flag == POOL_NOWAIT) {
            pthread_mutex_unlock(&(pool->mutex));
            return POOL_FULL;
        } else {
        // 대기열이 꽉 차서 더 이상 작업을 넣을 수 없는 경우 대기
        //while (pool->q_len == pool->q_size) {
            pthread_cond_wait(&(pool->full), &(pool->mutex));
        }
    }

    // 사용자가 요청한 작업을 대기열에 넣는다
    pool->q[(pool->q_len + pool->q_front) % (pool->q_size)].function = f;
    pool->q[(pool->q_len + pool->q_front) % (pool->q_size)].param = p;

    // 대기열 정보를 업데이트 해준다.
    (pool->q_len)++;

    // 대기열에 작업을 제출했으니 empty 조건 변수를 깨워준다.
    pthread_cond_signal(&(pool->empty));

    // lock을 해제한다.
    pthread_mutex_unlock(&(pool->mutex));

    // POOL_SUCCESS를 리턴한다
    return POOL_SUCCESS;
}
```

Pthread_pool_submit 함수

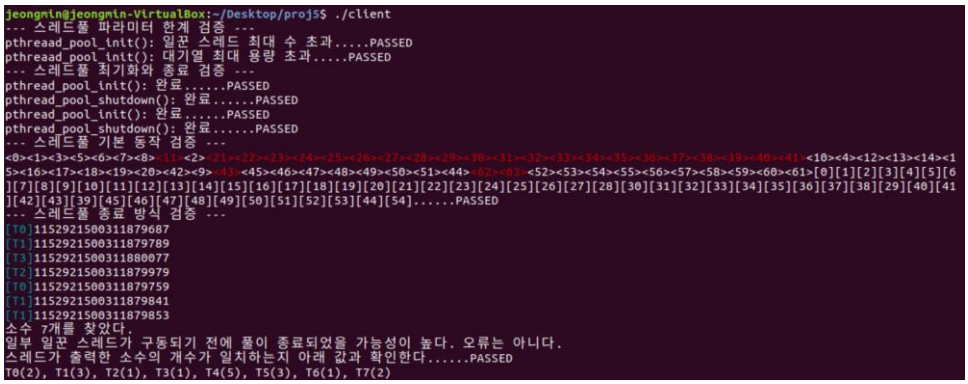
```c
int pthread_pool_shutdown(pthread_pool_t *pool, int how)
{
    pthread_mutex_lock(&(pool->mutex));

    // how에 따라 작업을 다 끝낼지 말지 결정
    if (how == POOL_COMPLETE) {
        // 대기열이 비어질 때까지 대기
        while (pool->q_len > 0) {
            pthread_cond_wait(&(pool->full), &(pool->mutex));
        }
    } else if (how == POOL_DISCARD) {
        // 대기열 비우기
        pool->q_front = 0;
        pool->q_len = 0;
    }
    pool->running = false;S
    pthread_cond_broadcast(&(pool->empty));
    pthread_cond_broadcast(&(pool->full));
    pthread_mutex_unlock(&(pool->mutex));

    for (int i = 0; i < pool->bee_size; i++) {
        pthread_join(pool->bee[i], NULL);
    }

    free(pool->q);
    free(pool->bee);

    pthread_mutex_destroy(&(pool->mutex));
    pthread_cond_destroy(&(pool->full));
    pthread_cond_destroy(&(pool->empty));

    return POOL_SUCCESS;
}
```

Pthread_pool_shutdown 함수

컴파일 과정을 보여주는 화면 캡처

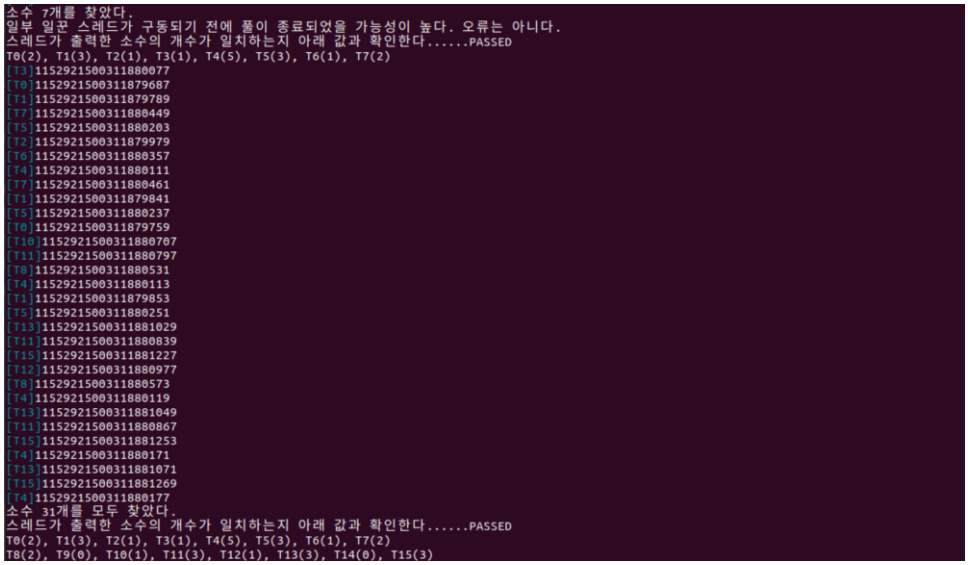


실행 결과물의 주요 장면을 발췌해서 그에 대한 상세한 설명




일꾼 스레드가 최대수를 초과하는지 대기열이 최대 용량을 초과하는지와 스레드풀 기본 동작을 검증하는 결과

스레드풀 종료 방식 검증에서는 DISCARD 일 때 검증하는데 소수가 6개 나왔다




그리고 COMPLETE일 때는 소수가 31개 나오면서 통과되었다.


그리고 마지막으로 무작위 검증 결과

## 과제를 수행하면서 경험한 문제점과 느낀 점

과제를 수행하면서 스레드풀이 무엇인지 이해하고, 스레드풀의 기능을 실제로 구현하면서 많은 것을 배울 수 있었습니다. 스레드풀은 다중 스레드 환경에서 작업을 효율적으로 관리하고 실행하는 데 사용되는 소프트웨어 디자인 패턴입니다.

스레드풀은 작업 큐 관리, 스레드 관리, 스레드 동기화, 작업 완료 통지 등의 기능을 제공합니다. 작업 큐는 작업을 저장하고 스레드가 가져와 처리할 수 있도록 합니다. 스레드풀은 미리 생성된 스레드 집합을 유지하며, 작업이 도착하면 사용 가능한 스레드에게 작업을 할당하고 작업이 완료되면 스레드를 재사용합니다.

동기화 메커니즘을 사용하여 여러 스레드의 동시 액세스를 관리하며, 일반적으로 뮤텍스나 세마포어 등을 활용합니다. 이를 통해 작업 큐에 대한 상호 배제와 스레드 간의 조율이 가능합니다. 작업이 완료되면 스레드풀은 작업 완료 통지를 제공하여 작업을 요청한 스레드가 결과를 받거나 작업이 완료되었음을 알 수 있게 합니다.

스레드풀을 구현하면서 작업 큐의 동시 액세스 관리, 작업 할당 및 완료 추적, 스레드 생성 및 삭제의 최소화 등 다양한 측면을 배울 수 있었습니다. 또한 스레드풀을 사용함으로써 작업 처리의 효율성과 성능을 향상시킬 수 있는 경험을 얻게 되었습니다. 스레드풀은 병렬 처리가 필요한 다양한 응용 프로그램에서 유용하게 활용될 수 있으며, 멀티스레드 환경에서의 개발과 성능 최적화에 대한 이해를 높이는 데 도움이 됩니다.