운영체제론 개인 과제 5 보고서

제출자 : 이찬영 (2019098068, ICT 융합학부)

1. 본인이 설계한 네 가지 버전에 대한 설명

<pthread_pool.c>

```c
#include "pthread_pool.h"
#include <stdlib.h>

/*
 * 풀에 있는 일꾼(일벌) 스레드가 수행할 함수이다.
 * FIFO 대기열에서 기다리고 있는 작업을 하나씩 꺼내서 실행한다.
 * 대기열에 작업이 없으면 새 작업이 들어올 때까지 기다린다.
 * 이 과정을 스레드풀이 종료될 때까지 반복한다.
 */
static void *worker(void *param)
{
    pthread_pool_t *pool = (pthread_pool_t *)param;     // 스레드풀 제어블록 생성

    while(1) {
        pthread_mutex_lock(&pool->mutex);      // 일꾼 스레드의 동기화를 위함

        // 대기열에 작업이 없고, 스레드풀이 실행 상태이면 빈 대기열에 새 작업이 들어올 때까지 기다림
        while(pool->q_len == 0 && pool->running == true) {
            pthread_cond_wait(&pool->full, &pool->mutex);
        }

        if(!pool->running) {    // 스레드풀이 종료되면, 반복문 탈출
          pthread_mutex_unlock(&(pool->mutex));
          break;
        }

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

        // 만약, 대기열의 자리가 하나 남아 있거나, 대기열의 길이가 0이면, &pool->empty에 시그널 보냄
        if(pool->q_len == pool->q_size-1 || pool->q_len == 0) {
          pthread_cond_signal(&pool->empty);
        }

```

```c
        // task를 수행한다.
        task.function(task.param);
    }
    pthread_exit(NULL);
}

/*
 * 스레드풀을 생성한다. bee_size는 일꾼(일벌) 스레드의 개수이고, queue_size는 대기열의 용량이다.
 * bee_size는 POOL_MAXBSIZE를, queue_size는 POOL_MAXQSIZE를 넘을 수 없다.
 * 일꾼 스레드와 대기열에 필요한 공간을 할당하고 변수를 초기화한다.
 * 일꾼 스레드의 동기화를 위해 사용할 상호배타 락과 조건변수도 초기화한다.
 * 마지막 단계에서는 일꾼 스레드를 생성하여 각 스레드가 worker() 함수를 실행하게 한다.
 * 대기열로 사용할 원형 버퍼의 용량이 일꾼 스레드의 수보다 작으면 효율을 극대화할 수 없다.
 * 이런 경우 사용자가 요청한 queue_size를 bee_size로 상향 조정한다.
 * 성공하면 POOL_SUCCESS를, 실패하면 POOL_FAIL을 리턴한다.
 */
int pthread_pool_init(pthread_pool_t *pool, size_t bee_size, size_t queue_size)
{
    // 만약 queue_size 나 일꾼 스레드의 개수가 최대치를 초과하면 POOL_FAIL을 리턴
    if (queue_size > POOL_MAXQSIZE || bee_size > POOL_MAXBSIZE) {
        return POOL_FAIL;
    }

    if (bee_size > queue_size) {
        queue_size = bee_size;
    }

    // 일꾼 스레드와 대기열에 필요한 공간 할당
    pool->bee = (pthread_t *)malloc(bee_size * sizeof(pthread_t));
    pool->q = (task_t *)malloc(queue_size * sizeof(task_t));

    if (pool->bee == NULL || pool->q == NULL) {     // 일꾼 스레드 배열이 없거나 원형 버퍼가 없을 때
        return POOL_FAIL;
    }

```

```c
    // 변수 초기화
    pool->running = true;
    pool->q_size = queue_size;
    pool->q_front = 0;
    pool->q_len = 0;
    pool->bee_size = bee_size;

    // 동기화를 위한 상호배타 락 및 조건변수 초기화
    pthread_mutex_init(&pool->mutex, NULL);
    pthread_cond_init(&pool->full, NULL);
    pthread_cond_init(&pool->empty, NULL);

    // 일꾼 스레드 생성
    int i;
    for (i = 0; i < bee_size; ++i) {
        pthread_create(&pool->bee[i], NULL, worker, (void*)pool);
    }

    return POOL_SUCCESS;    // 작업 요청이 성공하면 리턴
}

/*
 * 스레드풀에서 실행시킬 함수와 인자의 주소를 넘겨주며 작업을 요청한다.
 * 스레드풀의 대기열이 꽉 찬 상황에서 flag이 POOL_NOWAIT이면 즉시 POOL_FULL을 리턴한다.
 * POOL_WAIT이면 대기열에 빈 자리가 나올 때까지 기다렸다가 넣고 나온다.
 * 작업 요청이 성공하면 POOL_SUCCESS를 리턴한다.
 */
int pthread_pool_submit(pthread_pool_t *pool, void (*f)(void *p), void *p, int flag)
{
    pthread_mutex_lock(&pool->mutex);

    // 대기열이 꽉 찬 상황
    if (pool->q_len == pool->q_size) {
      if (flag == POOL_NOWAIT) {
        pthread_mutex_unlock(&pool->mutex);
        return POOL_FULL;     // 즉시 리턴
      }
      if (flag == POOL_WAIT) {
        while (pool->q_len == pool->q_size) {
          pthread_cond_wait(&pool->empty, &pool->mutex);    // 대기열에 빈 자리가 나올 때까지 기다렸다가 넣고 나옴
        }
      }
    }

    // 작업을 큐에 추가하고 작업을 요청한다.
    pool->q[(pool->q_front + pool->q_len) % pool->q_size].function = f;
    pool->q[(pool->q_front + pool->q_len) % pool->q_size].param = p;
    pool->q_len++;

    // 작업이 추가되면 새 작업이 들어오길 기다리고 있는 스레드에게 신호를 보낸다.
    pthread_cond_signal(&pool->full);
    pthread_mutex_unlock(&pool->mutex);

    return POOL_SUCCESS;    // 작업 요청이 성공하면 리턴
}

/*
 * 스레드풀을 종료한다. 일꾼 스레드가 현재 작업 중이면 그 작업을 마치게 한다.
 * how의 값이 POOL_COMPLETE이면 대기열에 남아 있는 모든 작업을 마치고 종료한다.
 * POOL_DISCARD이면 대기열에 새 작업이 남아 있어도 더 이상 수행하지 않고 종료한다.
 * 부모 스레드는 종료된 일꾼 스레드와 조인한 후에 스레드풀에 할당된 자원을 반납한다.
 * 스레드를 종료시키기 위해 철회를 생각할 수 있으나 바람직하지 않다.
 * 락을 소유한 스레드를 중간에 철회하면 교착상태가 발생하기 쉽기 때문이다.
 * 종료가 완료되면 POOL_SUCCESS를 리턴한다.
 */
int pthread_pool_shutdown(pthread_pool_t *pool, int how)
{
    // 일꾼 스레드가 현재 작업 중이면 대기열에 남아 있는 모든 작업을 마칠 때까지 기다림
    if (how == POOL_COMPLETE) {
        // 큐가 비어질 때까지 기다린다.
        pthread_mutex_lock(&pool->mutex);
        while (pool->q_len > 0) {
            pthread_cond_wait(&pool->empty, &pool->mutex);
        }
        pthread_mutex_unlock(&pool->mutex);
    }
    pool->running = false;      // 더 이상 수행하지 않고 종료함
    pthread_cond_broadcast(&pool->full);    // pool->full 조건변수에 신호를 보냄

    // 모든 일꾼 스레드를 종료시킨다.
    int i;
    for (i = 0; i < pool->bee_size; ++i) {
        pthread_join(pool->bee[i], NULL);
    }

    // 할당된 자원을 반납한다.
    free(pool->bee);
    free(pool->q);

    // 뮤텍스와 조건변수를 destroy
    pthread_mutex_destroy(&pool->mutex);
    pthread_cond_destroy(&pool->full);
    pthread_cond_destroy(&pool->empty);

    return POOL_SUCCESS;    // 작업 요청이 성공하면 리턴
}
```

worker 함수 : 일꾼 스레드를 수행하는 함수로, 스레드가 하나씩 실행될 수 있도록 mutex 락을 걸어줍니다. 스레드가 들어왔을 때, 대기열에 작업이 없고, 스레드풀이 실행 상태이면, 빈 대기열에 새 작업(task)이 들어올 때까지 조건변수 pool->full에서 기다리게 됩니다. (중간에, 스레드 풀이 종료되면 mutex를 해제하고 반복문을 탈출하게 되는 부분이 있는데 이것은 pthread_pool_shutdown() 함수와 연동됩니다.) 이후에는 대기열로부터 작업(task)을 얻습니다. 만약 대기열의 자리가 하나 생기거나, 대기열의 길이가 0이면, 빈 자리가 발생할 때까지 기다리는 조건변수 pool->empty에게 신호를 보냅니다. (대기열에 빈 자리가 발생할 때까지 기다리는 것은 pthread_pool_submit() 함수와 연동됩니다.) 그 다음에는 뮤텍스 락을 해제해 주고 난 뒤에 task를 수행합니다.

Pthread_pool_init 함수 : 스레드풀을 생성하는 함수로, 풀의 이름과 스레드의 수, 대기열 크기를 인자로 받습니다. 만약 queue_size와 bee_size는 시스템이 정한 최대치를 넘으면 POOL_FAIL을 리턴하며, 대기열의 용량이 일꾼 스레드 수보다 작으면, 일꾼 스레드 수와 대기열 크기가 같도록 만들어줍니다. 이후에는 일꾼 스레드와 대기열에 필요한 공간을 malloc()을 이용해 할당해 준 뒤, 스레드 제어 블록에 사용될 변수들을 초기화 해 주고 동기화를 위한 상호 배타 mutex 락, 조건변수 초기화 및 일꾼 스레드를 생성합니다. 작업 요청이 성공적으로 끝나면, POOL_SUCCESS를 리턴합니다.

Pthread_pool_submit 함수 : 스레드풀에 작업을 제출하는 함수로, 실행시킬 함수와 함수 인자, 대기열에 빈 자리가 생길 때까지 기다릴 것인지의 여부를 flag 라는 인자로 받습니다. 스레드가 하나씩 실행되게 하기 위해 mutex 락을 걸어줍니다. 대기열이 꽉 차지 않은 상황이면, flag == POOL_NOWAIT과 flag == POOL_FULL은 동일합니다. 하지만 대기열이 꽉 찬 상황에서 flag == POOL_NOWAIT 이면, mutex 락 해제와 함께 POOL_FAIL을 즉시 리턴하게 되며, flag == POOL_WAIT 이면, 대기열에 빈자리가 나올 때까지 조건변수 pool->empty에서 기다리게 됩니다. 이후에는 작업을 큐에 추가하고, 작업을 요청하게 됩니다. 작업이 추가되면 새 작업이 들어오길 기다리고 있는 조건변수 pool->full에게 신호를 보낸 뒤, mutex 락을 해제합니다. 작업 요청이 성공적으로 끝나면, POOL_SUCCESS를 리턴합니다.

Pthread_pool_shutdown() 함수 : 스레드 풀을 종료하는 함수이며, 풀의 이름과 대기열에 남아 있는 작업을 다 끝내고 종료할 것인지의 여부를 물어보는 how를 인수로 받습니다. 만약, how == POOL_COMPLETE 이면, 대기열에 남아 있는 모든 작업이 끝날 때까지 기다리게 됩니다. how == POOL_DISCARD 이면, 대기열에 새 작업이 남아 있어도 더 이상 수행하지 않고 바로 종료시키며, 새 작업이 들어올 때까지 기다리는 조건변수 pool->full에 있는 모든 스레드에게 신호를 보냅니다. 따라서 모든 스레드들은 대기열로부터 나오게 되고, pthread_join에 의해 성공적으로 종료됩니다.

이후에는 할당된 자원을 반납하고, mutex와 조건변수를 destroy 합니다. 작업 요청이 성공적으로 끝나면, POOL_SUCCESS를 리턴합니다.

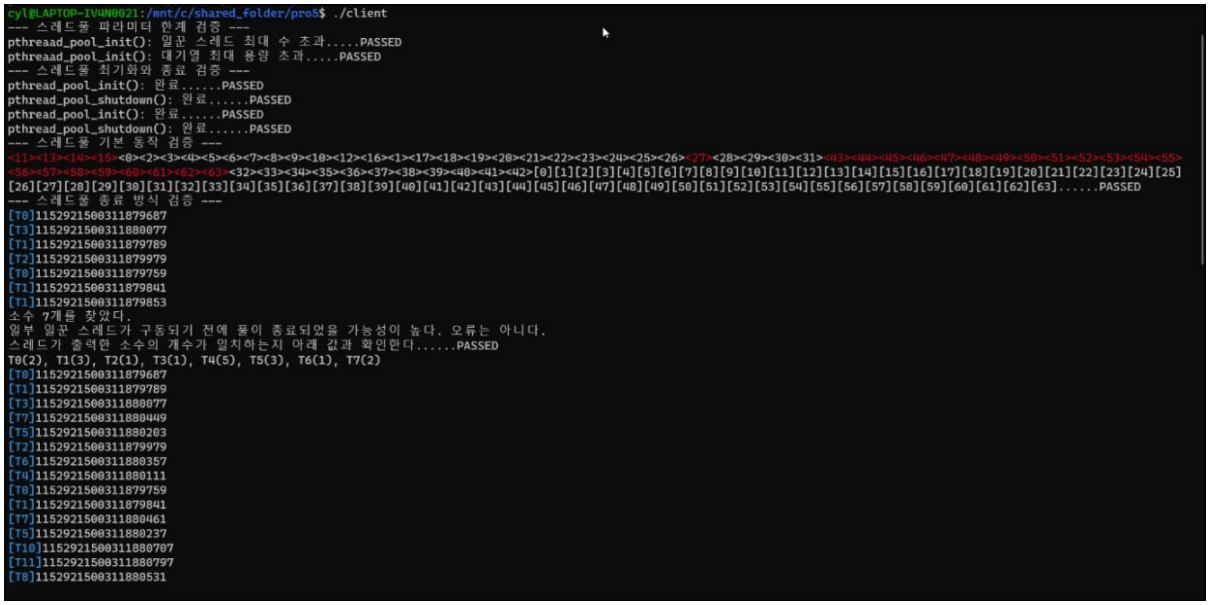
<pthread_pool.h>

따로 수정을 하지는 않았습니다.

2. 컴파일 과정을 보여주는 화면 캡처

아무 오류 없이 pthread_pool.c가 컴파일이 잘 된 것을 확인할 수 있습니다.

3. 실행 결과물의 주요 장면을 발췌해서 그에 대한 상세한 설명



스레드풀 검증이 예상대로 잘 되는 것을 알 수 있습니다. 스레드풀 종료 방식 검증에서는, 검사되는 소수의 개수가 매번 달라집니다. 하지만 소수 31개를 모두 찾습니다. 여러 시스템에서 프로그램을 실행해 본 결과 시스템 차이로 인한 실행 시간 차이는 존재했습니다.

4. 과제를 수행하면서 경험한 문제점과 느낀 점

이번 과제를 수행하면서 스레드풀과 그것을 실행하는 방식에 대해서 배울 수 있었습니다. 스레드 풀이란, 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리하는 것을 말합니다. 이것은 매번 스레드를 생성/수거 하는데 드는 프로그램에 따르는 부담을 줄이기 위한 것이며, 다수의 사용자 요청을 처리하고 대응하기 위함입니다.

과제를 수행하는 중, 스레드풀에 대한 개념이 이해가 잘 가지 않아서 고민하는 시간이 필요했던 것 같습니다. 그 결과, 스레드풀뿐만 아니라, 작업 대기열에 들어오는 task도 함께 고려해야 한다는 것을 깨닫게 되었습니다. 또한 이전에 배웠던 뮤텍스 락과 조건변수가 스레드 풀을 구현하는데 쓰이며, 이것은 생각보다 실행에 있어서 많은 영향을 끼친다는 것을 알 수 있었습니다.

또한, 구조체와 포인터 접근에 대한 이해가 부족했었는데, 이번 기회에 복습하고 그 쓰임새에 대해서 확실히 알 수 있어서 좋았습니다. 동적할당에 대해서는 malloc으로 공간을 할당해 주었으면, 중간에 그 공간을 추월하면 안 되는 것과 동시에 해제도 잘 해주어야 한다는 것을 알 수 있었습니다. Segmentation fault 오류가 종종 발생했는데, 이 부분은 포인터 참조를 잘못해서 발생한다는 것을 수업 시간 때 교수님이 알려주셨습니다.

과제 5개를 모두 수행하면서 과제 하나하나가 생각을 많이 하게 하고, 각각의 개념들에 대해서 완벽한 이해를 요구한다는 생각을 하게 되었습니다. 확실히 수업과 과제로부터 얻은 것들이 많아서 뿌듯합니다.

남은 기말고사도 성실히 임하겠습니다. 감사합니다.