# 스레드풀

## 1. 컴파일 과정

```
(base) parkminseon@172 ~/Documents/3-1/OS/assignment/proj5 make
gcc -Wall -O -c client.c
gcc -Wall -O -c pthread_pool.c
gcc -o client client.o pthread_pool.o
(base) parkminseon@172 ~/Documents/3-1/OS/assignment/proj5 ./client
```

## 2. 코드 설명

- **pthread_pool_init**

```c
int pthread_pool_init(pthread_pool_t *pool, size_t bee_size, size_t queue_size)
{
    // ? 스레드 수와 버퍼 크기가 최대 용량을 넘을 경우 POOL_FAIL을 리턴 후 종료
    if (bee_size > POOL_MAXBSIZE || queue_size > POOL_MAXQSIZE)
        return POOL_FAIL;

    // ? 원형 버퍼가 스레드 수보다 작으면 버퍼 크기를 스레드 수로 상향조정
    if (queue_size < bee_size)
        queue_size = bee_size;

    // ? bee_size, q_size 값 할당
    pool->bee_size = bee_size;
    pool->q_size = queue_size;
    // ? q, bee 배열 초기화 및 할당
    pool->q = calloc(queue_size, sizeof(task_t)); // 스택 영역은 함수가 종료되면서 값이 소멸되기 때문
    pool->bee = calloc(bee_size, sizeof(pthread_t));

    // ? q_front, q_len 값을 0으로 초기화
    pool->q_front = 0;
    pool->q_len = 0;

    // ? 뮤텍스 초기화
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);
    pool->mutex = mutex;

    // ? 조건변수 초기화
    pthread_cond_t full, empty;
    pthread_cond_init(&full, NULL);
    pthread_cond_init(&empty, NULL);
    pool->full = full;
    pool->empty = empty;

    // ? 스레드풀 실행 상태를 true로 설정
    pool->running = true;
    pool->is_pool_complete = false;

    // ? 일꾼 스레드 생성 -> worker 실행하도록
    for (int i = 0; i < bee_size; ++i) {
        pthread_create(pool->bee+i, NULL, worker, pool);
    }
    return POOL_SUCCESS;
}
```

`pthread_pool_init`은 스레드 풀을 초기화하는 함수입니다. 따라서 인자로 전달 받은 `pool 구조체`에 필요한 자원을 할당하고, 초기화 과정을 성공적으로 완료하면 `POOL_SUCCESS`를 그렇지 않으면 `POOL_FAIL`을 리턴합니다. 그리고 만약 전달 받은 **스레드 수**와 **대기열 크기**가 최대 용량을 넘어서면 초기화를 진행할 수 없도록 바로 `POOL_FAIL`을 반환하였고, **대기열 크기**가 **스레드 수** 보다 작으면 효율적인 작업이 어렵기 때문에 이를 완화하기 위해 대기열 크기를 스레드 수만큼 상향조정 하였습니다.

여기서 주목해야 할 부분은 **대기열 배열**과 **일꾼 스레드 배열**을 calloc(혹은 malloc)을 사용하여 `힙 영역`에 할당했다는 점입니다. 만약 calloc을 사용하여 힙 영역에 직접 할당하지 않고 곧바로 배열로 값을 할당하게 되면 스택 영역에 공간에 할당되는데, 이렇게 되면 **init 함수가 종료되는 순간** 해당 함수가 차지한 **스택 영역도 소멸**되기 때문에 배열 공간을 제대로 할당받을 수 없게 됩니다. 따라서 이 부분은 calloc 함수를 사용하여 구현하였습니다.

그리고 기존의 `pthread_pool_t` 구조체에는 없던 **is_pool_complete**을 추가했습니다. 이는 스레드풀이 `POOL_COMPLETE` 옵션으로 종료되는 경우를 처리하기 위해 추가한 변수로, **pthread_pool_shutdown** 함수 구현 부분에서 설명을 덧붙이도록 하겠습니다.

이렇게 pool 구조체의 값을 모두 정상적으로 설정한 후에는 앞서 생성한 일꾼 스레드들에게 작업을 할당한 후 `POOL_SUCCESS`를 리턴할 수 있습니다.

- **pthread_pool_submit**

```c
int pthread_pool_submit(pthread_pool_t *pool, void (*f)(void *p), void *p, int flag)
{
    /*
    pool:   스레드풀
    f:      작업을 수행할 함수 포인터
    p:      f 함수에 넘겨주는 인자 포인터
    flag:   대기열이 꽉 찼을 때 기다릴 지 여부 (POOL_WAIT / POOL_NOWAIT)
    */
    // ? 대기열을 조회하기 전 스레드 풀의 mutex에 락을 건다
    pthread_mutex_lock(&(pool->mutex));

    // ? 대기열이 꽉 찬 경우
    if (pool->q_len == pool->q_size) {
        // ? 대기하지 않는다면 POOL_FULL 리턴하여 종료
        if (flag == POOL_NOWAIT) {
            pthread_mutex_unlock(&(pool->mutex));
            return POOL_FULL;
        }
        // ? 대기를 원한다면 shutdown에서 스레드 풀을 종료하기 전까지, 대기열이 빌 동안 대기한다.
        while (pool->running && pool->q_len == pool->q_size)
            pthread_cond_wait(&(pool->empty), &(pool->mutex));
    }

    // ? 대기열이 빌 때까지 대기하는 도중에 스레드 풀이 종료된 경우, 대기열에 넣지 않고 종료시킨다.
    if (!pool->running) {
        pthread_mutex_unlock(&(pool->mutex));
        return POOL_FULL;
    }

    // ? 작업 생성 후 스레드풀의 대기열에 작업을 추가한다.
    task_t task;
    task.function = f;
    task.param = p;
    int input_index = (pool->q_front + pool->q_len) % (pool->q_size);
    pool->q[input_index] = task;
    pool->q_len++; // 대기열에 들어온 작업의 수를 1 증가시킨다.

    pthread_cond_signal(&(pool->full)); // 작업 생성을 기다리는 일꾼 스레드를 깨운다.
    pthread_mutex_unlock(&(pool->mutex));
    return POOL_SUCCESS;
}
```

이 함수는 *스레드풀 대기열에 새로운 작업을 추가하는 함수*입니다. 우선 대기열에 작업을 추가할 수 있는지 확인하기 위해 **mutex 락**을 얻어야 합니다. 락을 얻은 후에는 **대기열이 꽉 찼는지 검사**합니다. `POOL_NOWAIT` 옵션인 경우에는 *대기열이 빌 때까지 대기하지 않고 곧바로 종료*시켜야하기 때문에, **mutex 락을 해제**한 후 `POOL_FULL`을 리턴하여 함수를 종료합니다.

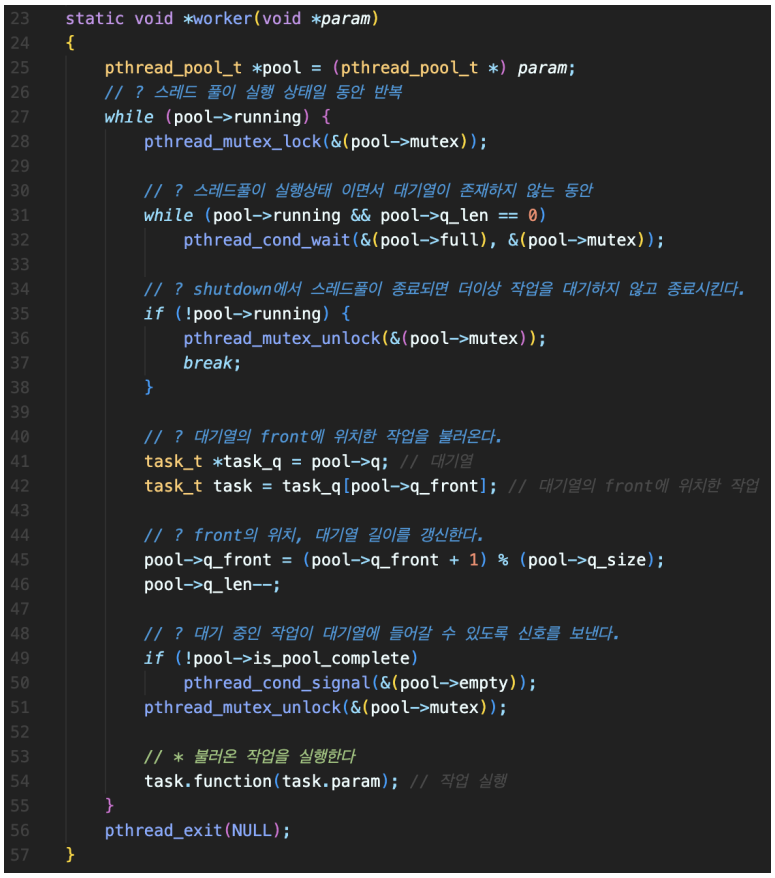
만약 그렇지 않은 경우 *대기열이 빌 때까지 대기해야* 하므로, **일꾼 스레드**가 작업 하나를 꺼낸 후 ***signal***을 보낼 때까지 `empty 조건변수`에서 대기합니다.

따라서 대기열에 공간이 생길 때까지 `empty 조건변수`에서 대기하다가 **일꾼 스레드**가 신호를 보내면 대기를 멈춥니다. 이때 스레드가 mutex 락을 얻는 사이에 다른 스레드에 의해 조건이 변할 수 있기 때문에, while루프로 조건을 재확인할 수 있도록 했습니다. 그리고 empty 조건변수에서 대기하는 동안 **shutdown 함수**로 인해 스레드풀이 종료(running 값이 false로 변경)될 가능성이 있기 때문에 ***스레드풀의 running 값***도 함께 확인하도록 구현했습니다.

이후 작업을 대기열에 추가하기 전, 스레드풀이 아직 실행중인지 확인합니다. 스레드풀이 종료된 상태라면 mutex 락을 해제하고 함수를 종료합니다. 이때의 **리턴 값**은 *대기열이 꽉 차서 대기하다가 스레드풀이 종료되어 대기열에 포함되지 못한 경우*이기 때문에 `POOL_FULL` 옵션으로 설정했습니다.

작업을 추가한 후에는 대기열에 새로운 작업이 들어오는 것을 `full 조건변수`에서 대기하는 일꾼 스레드가 있을 수 있기 때문에 이를 깨웁니다. 이후 mutex 락을 해제시킨 후 함수를 종료합니다.

- **worker (일꾼 스레드)**



```c
static void *worker(void *param)
{
    pthread_pool_t *pool = (pthread_pool_t *) param;
    // ? 스레드 풀이 실행 상태일 동안 반복
    while (pool->running) {
        pthread_mutex_lock(&(pool->mutex));

        // ? 스레드풀이 실행상태 이면서 대기열이 존재하지 않는 동안
        while (pool->running && pool->q_len == 0)
            pthread_cond_wait(&(pool->full), &(pool->mutex));

        // ? shutdown에서 스레드풀이 종료되면 더이상 작업을 대기하지 않고 종료시킨다.
        if (!pool->running) {
            pthread_mutex_unlock(&(pool->mutex));
            break;
        }

        // ? 대기열의 front에 위치한 작업을 불러온다.
        task_t *task_q = pool->q; // 대기열
        task_t task = task_q[pool->q_front]; // 대기열의 front에 위치한 작업

        // ? front의 위치, 대기열 길이를 갱신한다.
        pool->q_front = (pool->q_front + 1) % (pool->q_size);
        pool->q_len--;

        // ? 대기 중인 작업이 대기열에 들어갈 수 있도록 신호를 보낸다.
        if (!pool->is_pool_complete)
            pthread_cond_signal(&(pool->empty));
        pthread_mutex_unlock(&(pool->mutex));

        // * 불러온 작업을 실행한다
        task.function(task.param); // 작업 실행
    }
    pthread_exit(NULL);
}
```

이 함수는 일꾼 스레드가 실행하는 함수로, 대기열에 있는 작업을 하나씩 꺼내어 실행하는 역할을 합니다. 따라서 작업을 꺼내기 전 대기열에 작업이 존재하는지 확인하기 위해 mutex 락을 얻습니다. submit 함수와 유사하게 작업이 하나라도 생성될 때까지 full 조건변수에서 대기합니다. 마찬가지로 full 조건변수에서 대기하는 도중에 스레드풀이 종료될 수 있으므로 이를 같이 확인합니다.

이후 `q_front`가 가리키는 작업을 꺼내고, 이 작업을 실행하기 전에 `q_len`을 감소시킵니다. 그리고 `empty 조건변수`에서 대기중인 submit 함수가 대기열에 작업을 새로 추가할 수 있도록 ***signal***을 보내는데, shutdown 함수에서 `POOL_COMPLETE` 옵션으로 스레드풀을 종료시키기 위해 ***대기열의 모든 작업이 완료되기를 기다리는 경우***에는 이 신호를 보내지 않도록 했습니다. `POOL_COMPLETE`인 경우에는 *대기열에 남아있는 작업만* 실행해야 합니다. 즉, 조건변수에서 대기중인 작업들은 더이상 대기열에 추가되면 안되기 때문에 이를 위와 같이 처리한 것입니다.

일꾼 스레드는 앞서 꺼내온 작업을 실행하기 위해 mutex 락을 해제한 후 작업을 시작합니다. 그리고 **shutdown 함수**에서 `POOL_COMPLETE` 옵션으로 스레드풀을 종료시킨 경우, 대기열에 있는 모든 작업이 끝나면 *running값이 false*가 되면서 일꾼 스레드가 while 루프를 빠져나올 수 있게 설계했습니다.

- **pthread_pool_shutdown**

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

    // ? POOL_COMPLETE인 경우
    if (how == POOL_COMPLETE) {
        pool->is_pool_complete = true;
        // ? 대기열에 남아있는 모든 작업을 마칠 때까지 대기한다.
        while (pool->q_len > 0) {
            pthread_mutex_unlock(&(pool->mutex));
            pthread_mutex_lock(&(pool->mutex));
        }
    }

    // ? 조건변수에서 대기중인 스레드, 작업을 진행중인 스레드가 더이상 작업을 이어가지 않도록 running을 false로 설정
    pool->running = false;

    // ? 대기열이 비어있는 상태에서 shutdown이 호출되면, full cond에서 작업 할당을 기다리고 있는 일꾼 스레드가 존재
    pthread_cond_broadcast(&(pool->full));
    pthread_cond_broadcast(&(pool->empty));

    pthread_mutex_unlock(&(pool->mutex));

    // ? 모든 일꾼 스레드가 종료될 때까지 대기한다.
    for (int i = 0; i < pool->bee_size; ++i) {
        pthread_join(*(pool->bee+i), NULL);
    }

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

    return POOL_SUCCESS;
}
```

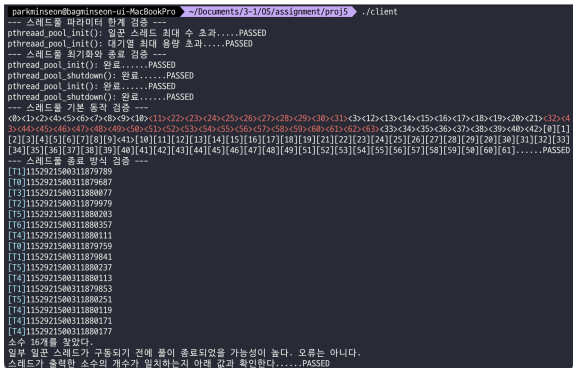
이 함수는 스레드 풀을 종료시키는 함수입니다. shutdown 함수도 마찬가지로 스레드 풀의 상태를 확인하기 위해 mutex 락을 얻습니다.

종료 옵션이 `POOL_COMPLETE`인 경우에는 *대기열에 남아있는 작업이 모두 종료될 때까지 대기*해야 합니다. 여기서 특히 주의해서 구현했던 부분은 **running을 false 값으로 변경하는 구간**입니다. 곧바로 running을 false로 바꾸게 되면 일꾼 스레드가 더이상 남은 작업을 더 수행할 수 없기 때문에, **대기열의 모든 작업을 끝낼 때까지 running 값을 true로 유지시켜야** 합니다. 따라서 while문을 사용하여 대기열의 길이가 0이 될 때까지 기다릴 수 있도록 했습니다. 락을 얻은 시점에 아직 대기열이 남아있다면 락을 해제하고 다시 락을 기다리게 됩니다.

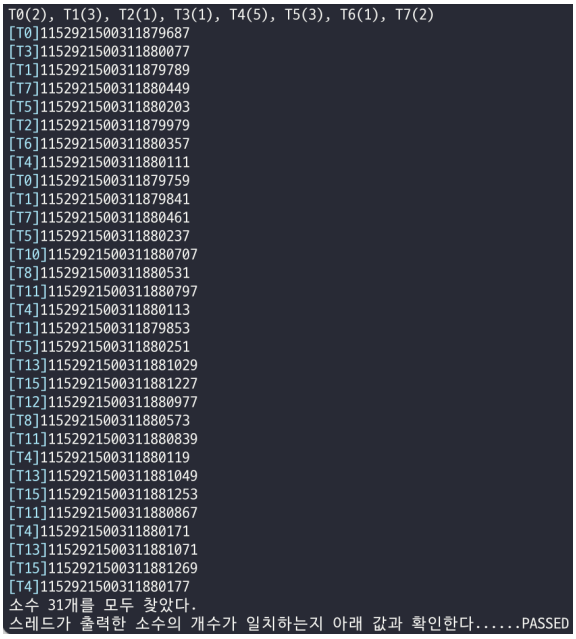
*일꾼 스레드가 대기열에 남은 작업을 모두 처리*했거나 *종료 옵션이* `POOL_COMPLETE` *이 아닌 경우*(= `POOL_DISCARD`), **running을 false로 변경**하여 일꾼 스레드가 더이상 작업을 진행하지 못하도록 했습니다. 그리고 조건변수에 대기중인 스레드가 남아있을 수 있기 때문에 이를 모두 깨우도록 **broadcast**를 호출했습니다.

마지막으로 아직 작업을 미처 마치지 못한 일꾼 스레드가 있을 수 있기 때문에 이를 기다리기 위해 **join**하였고, 모든 일꾼 스레드가 종료되면 사용했던 `조건변수`와 `mutex`를 반납하기 위해 **destroy** 시킵니다. 그리고 bee 배열과 q 배열은 힙 영역에 할당하였기 때문에 이를 해제시키기 위해 free 함수도 사용하였습니다.

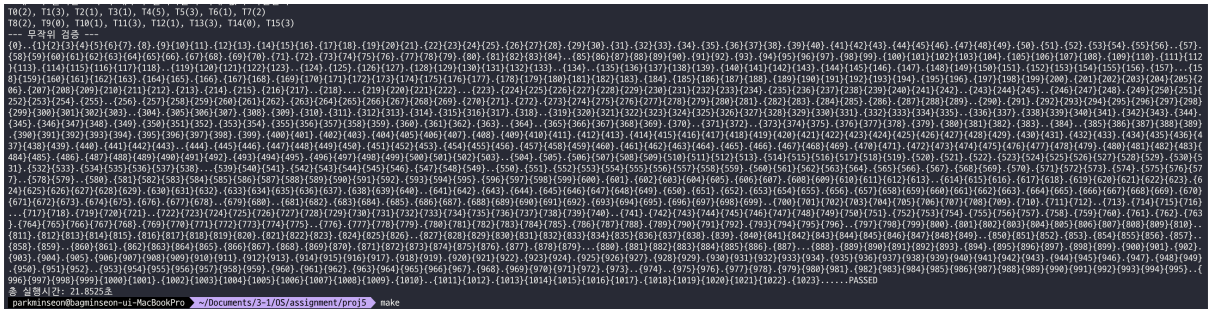
# 3. 실행 결과




1




2




3

첫 번째 사진은 return 값이 잘 반환되었는지 먼저 확인한 후, 스레드 풀이 잘 동작하는지 확인하는 테스트 코드의 결과를 보여주고 있습니다. **스레드풀 기본 동작 검증 과정**에서는 `POOL_NOWAIT` 옵션으로 작업을 추가한 경우, 대기열이 꽉 차있을 때는 곧바로 `POOL_FULL`을 반환하여 작업이 대기열에 추가되지 않습니다. 이런식으로 버려진 작업들이 빨간색으로 출력된 것을 사진에서 확인할 수 있습니다. 이후 진행된 **스레드풀 종료 방식 검증 과정**에서는 `POOL_DISCARD` 옵션으로 스레드풀을 종료시키기 때문에, 일찍이 일꾼 스레드로부터 선택받지 못한 작업들은 곧바로 버려지므로 모든 소수를 찾지 못한 것을 볼 수 있습니다.

두 번째 사진은 `POOL_COMPLETE` 옵션으로 스레드풀을 종료시킵니다. 이 옵션은 **대기열에 남아있는 작업까지 모두 끝낸 후 종료**하기 때문에 모든 소수를 찾은 것을 볼 수 있습니다.

세 번째 무작위 검증 과정에서는 submit 옵션, shutdown 옵션을 무작위로 선택하여 검증합니다. 이 과정에서도 데드락 없이 총 1024개의 숫자를 모두 정상적으로 출력하고 종료됨을 확인할 수 있습니다.

# 4. 문제점 및 느낀점

지금까지 배운 내용을 기반으로 코드를 전반적으로 구성하여 프로젝트를 진행할 수 있었습니다. 앞선 과제들은 뼈대코드가 주어지고 코드를 추가하는 방식이였다면, 이번 과제는 배운 내용을 종합하여 상황에 따라 필요한 내용을 적절하게 활용하는 것이 관건이였습니다.

우선 맨 처음 init 함수를 구현했었는데, 처음에는 함수 내에서 곧바로 배열로 값을 할당했다가 `segment fault` 오류를 마주쳤습니다. 해결 방법을 고민하다가 문득 **지역변수는 스택 영역에 생성되며 함수가 종료되면 소멸된다는 점**을 떠올릴 수 있었습니다. 이를 토대로 스택 영역에 할당하는 것이 아닌, 함수가 종료되어도 할당된 메모리 영역이 유지될 수 있도록 **힙 영역**에 배열 공간을 할당하는 것으로 해결책을 제시하였고, 이를 구현하는 과정에서 calloc(혹은 malloc) 함수도 활용해볼 수 있었습니다.

이후 나머지 코드들은 큰 막힘 없이 작성할 수 있었고, 이를 통해 mutex 락과 조건변수의 동작 흐름을 이전(과제4를 해결했을 때)보다 잘 이해하고 있음을 느낄 수 있었습니다. 다만 한 가지 아쉬운 부분은 **pthread_pool_shutdown** 함수에서 불필요하게 락을 얻을 수도 있다는 점입니다. while문으로 조건을 확인하면서 *조건이 성립하지 않으면 곧바로 락을 해제*시키

기 때문에, 코드가 다소 비효율적일 수 있다는 아쉬움이 있습니다. 시간이 부족하여 실제로 개선하진 못했지만, 스레드풀 구조체에 mutex 락을 추가하여 코드를 수정하면 조금 더 효율적으로 수정할 수 있을 것 같습니다.