운영체제 프로젝트 4

2017012106 손민우

알고리즘 설명

기존에 있던 변수 외에 환형 큐 이용을 위해 q_head, q_tail, q_current_size를 추가했습니다.

- void pool_init(void)

뮤텍스와 세마포를 각각 mutex, sem으로 초기화한다. 여기서 sem의 초깃값은 0이다. 꿀벌의 수)만큼 pthread_create로 스레드를 생성한다. 실행하는 worker 함수는 아래서 설명한다.

- void pool_shutdown(void)

꿀벌의 수만큼 생성했던 스레드를 pthread_cancel로 제거하고, pthread_join으로 모든 스레드 종료까지 기다리고, 세마포와 mutex를 제거한다.

- int pool_submit(void (*f)(void *p), void *p)

1. 구조체 new_work 생성 후 맴버인 function과 data를 받은 인자인 f와 p로 할당해준다.
2. new_work를 enqueue함수에 넣어줬을 때 0이 리턴된다면 정상적으로 큐에 삽입된 것이므로 sem_post로 sem세마포 값을 늘려주고 0을 리턴한다.
3. 1이 리턴된 경우에는 작업 요청이 실패했다는 뜻이므로 1을 리턴한다.

- static int enqueue(task_t t)

1. 환형 큐에 이용되는 공유변수를 이용하기 위해 mutex를 lock한다.
2. 환형 큐가 가득 차있는지 여부를 판별하기 위해 q_current_size가 QUEUE_SIZE와 같은지 비교하고, 만약 같다면 가득 차있다는 뜻이므로 잠근 mutex를 unlock한후 1을 리턴한다.
3. 가득 차있지 않을 경우 q_current_size에 1을 더하고 worktodo[q_tail]에 인자로 들어온 t를 저장한다.
4. q_tail을 인덱스로 바꿔줘야 하므로 q_tail을 (q_tail+1)%(QUEUE_SIZE)로 변경한다.
5. 모든 공유변수 이용이 끝났으므로 1에서 잠근 mutex를 unlock하고 0을 리턴한다.

- static int dequeue(task_t *t)

1. 환형 큐에 이용되는 공유변수를 이용하기 위해 mutex를 lock한다.
2. 환형 큐가 비어있는지 여부를 판별하기 위해 q_current_size가 0인지 체크한다. 만약 0이라면 대기열에 작업이 없다는 뜻이므로 바로 mutex를 unlock하고 1을 리턴한다.
3. 작업이 있을 경우 포인터 인자로 받은 t에 worktodo[q_head]를 할당해주고 q_head에는 (q_head+1)%(QUEUE_SIZE) 값을 넣어주어 큐의 맨 앞 작업을 pop해준다.
4. 모든 공유변수 이용이 끝났으므로 1에서 잠근 mutex를 unlock하고 0을 리턴한다.

- static void *worker(void *param)

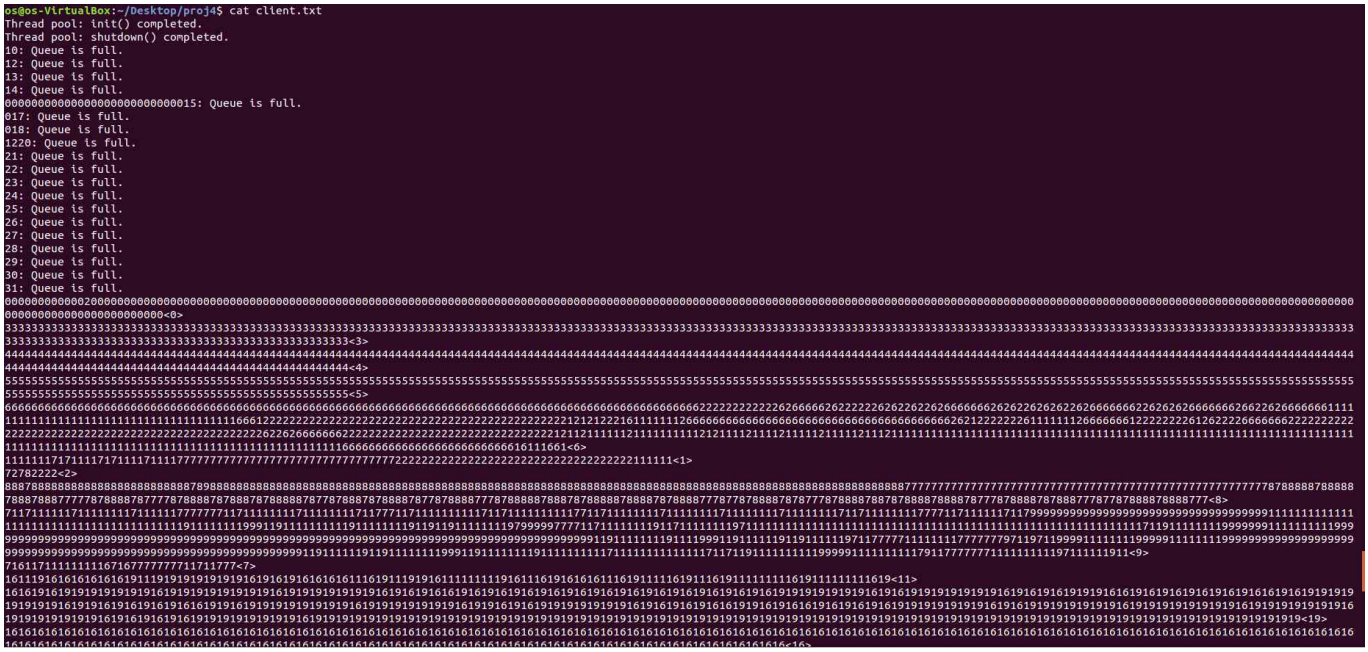
1. 실행할 작업을 담을 task_t형 구조체인 work를 생성한다.
2. 일꾼 스레드는 종료 전까지 계속해서 작업을 실행하므로 while(1)문을 사용한다.
3. 실행할 수 있는 작업 수는 한정되어있으므로 sem_wait로 sem 세마포 값을 줄인다.
4. work의 주소를 dequeue에 넣어줬을 때 0이 리턴되면 정상적으로 처리되었다는 뜻이므로 dequeue 함수 내에서 포인터 형식으로 받아온 function에 data를 넣고 실행시킨다.
5. static void함수는 리턴을 해주지 않으면 warning이 나오므로 param을 리턴해준다.

컴파일 과정

```
os@os-VirtualBox:~/Desktop/proj4$ make
gcc -Wall -c client.c -lpthread
gcc -Wall -c threadpool.c -lpthread
gcc -Wall -g3 -o client client.o threadpool.o -lpthread
os@os-VirtualBox:~/Desktop/proj4$ ./client > client.txt
```

실행 결과물

```
os@os-VirtualBox:~/Desktop/proj4$ cat client.txt
Thread pool: init() completed.
Thread pool: shutdown() completed.
10: Queue is full.
12: Queue is full.
13: Queue is full.
14: Queue is full.
00000000000000000000000015: Queue is full.
017: Queue is full.
018: Queue is full.
1220: Queue is full.
21: Queue is full.
22: Queue is full.
23: Queue is full.
24: Queue is full.
25: Queue is full.
26: Queue is full.
27: Queue is full.
28: Queue is full.
29: Queue is full.
30: Queue is full.
31: Queue is full.
```



```
1616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616161616<16>
hello
```

```
os@os-VirtualBox:~/Desktop/proj4$
```

설명

1. client.c 흐름대로 스레드풀 초기화와 종료 테스트를 한다.
2. 스레드풀이 가동되면서 거절된 요청들은 오류 메시지를 출력한다.
3. 0번째 작업이 실행되어 dequeue되고 11번 작업이 enqueue된 것을 알 수 있다.
4. 10번~15번까지는 큐에 적재되지 못하고 있다.
5. 1 or 2번째 작업이 실행되어 dequeue되고 16번 작업이 enqueue되었다.
6. 17,18번 작업은 거절되고 1or2번째 작업이 실행되어 19번째 작업이 enqueue되었다.
7. 모든 오류메시지 출력이 끝나고 2번째 작업이 시작된다. 그리고 스레드풀이 동작하면서 숫자들이 섞여서 출력되고 있다.
8. 숫자 출력 과정에선 한 스레드가 작업을 마치기 전까지 지정된 3개의 작업들이 스레드풀에서 수행되면서 한번에 3개 이하의 숫자들만 섞여서 나오는 것을 알 수 있다.
9. 그 후로 큐에 적재된 작업들이 출력되고 있고, 마지막에 적재된 16번 작업이 끝나면서 hello 문구가 나온다.
10. 마지막 작업은 아무런 방해가 없이 출력되므로 곰돌이가 정상적으로 출력된다.

소스 코드

```
os@os-VirtualBox:~/Desktop/proj4$ cat threadpool.c
/*
 * Copyright 2021. Heekuck Oh, all rights reserved
 * 이 프로그램은 한양대학교 ERICA 소프트웨어학부 재학생을 위한 교육용으로 제작되었습니다.
 */
#include <pthread.h>
#include <stdio.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
#include "threadpool.h"
#include <stdlib.h>
/*
 * 스레드 풀의 FIFO 대기열 길이와 일꾼 스레드의 갯수를 지정한다.
 */
#define QUEUE_SIZE 10
#define NUMBER_OF_BEES 3

/*
 * 스레드를 통해 실행할 작업 함수와 함수의 인자정보 구조체 타입
 */
typedef struct {
    void (*function)(void *p);
    void *data;
} task_t;

/*
 * 스레드 풀의 FIFO 대기열인 worktodo 배열로 원형 버퍼의 역할을 한다.
 */
static task_t worktodo[QUEUE_SIZE];
int q_head = 0;
int q_tail = 0;
int q_current_size = 0;
/*
 * mutex는 대기열을 조회하거나 변경하기 위해 사용하는 상호배타 락이다.
 */
static pthread_mutex_t mutex;

/*
 * 대기열에 새 작업을 넣는다.
 * enqueue()는 성공하면 0, 꽉 차서 넣을 수 없으면 1을 리턴한다.
 */
static int enqueue(task_t t)
{
    pthread_mutex_lock(&mutex);
    if(q_current_size == QUEUE_SIZE) {
        pthread_mutex_unlock(&mutex);
        return 1;
    }
```

```
    }
    q_current_size += 1;
    worktodo[q_tail] = t;
    q_tail = (q_tail + 1) % (QUEUE_SIZE);
    pthread_mutex_unlock(&mutex);
    return 0;
}

/*
 * 대기열에서 실행을 기다리는 작업을 꺼낸다.
 * dequeue()는 성공하면 0, 대기열에 작업이 없으면 1을 리턴한다.
 */
static int dequeue(task_t *t)
{
    pthread_mutex_lock(&mutex);
    if(q_current_size == 0) {
        pthread_mutex_unlock(&mutex);
        return 1;
    }
    q_current_size -= 1;
    *t = worktodo[q_head];
    q_head = (q_head + 1) % (QUEUE_SIZE);
    pthread_mutex_unlock(&mutex);
    return 0;
}

/*
 * bee는 작업을 수행하는 일꾼 스레드의 ID를 저장하는 배열이다.
 * 세마포 sem은 카운팅 세마포로 그 값은 대기열에 입력된 작업의 갯수를 나타낸다.
 */
static pthread_t bee[NUMBER_OF_BEES];
static sem_t sem;

/*
 * 풀에 있는 일꾼 스레드로 FIFO 대기열에서 기다리고 있는 작업을 하나씩 꺼내서 실행한다.
 */
static void *worker(void *param)
{
    task_t work;
    while(1) {
        sem_wait(&sem);
        if(dequeue(&work) == 0) {
                (*work.function)(work.data);
        }
    }
    return param;
}
```

```
/*
 * 스레드 풀에서 실행시킬 함수와 인자의 주소를 넘겨주며 작업을 요청한다.
 * pool_submit()은 작업 요청이 성공하면 0을, 그렇지 않으면 1을 리턴한다.
 */
int pool_submit(void (*f)(void *p), void *p)
{
    task_t new_work;
    new_work.function = f;
    new_work.data = p;
    if (enqueue(new_work) == 0) {
        sem_post(&sem);
        return 0;
    }
    return 1;
}

/*
 * 각종 변수, 락, 세마포, 일꾼 스레드 생성 등 스레드 풀을 초기화한다.
 */
void pool_init(void)
{
    pthread_mutex_init(&mutex, NULL);
    sem_init(&sem, 0, 0);
    for(int i = 0; i < NUMBER_OF_BEES; ++i)
        pthread_create(&bee[i], NULL, worker, NULL);
}

/*
 * 현재 진행 중인 모든 일꾼 스레드를 종료시키고, 락과 세마포를 제거한다.
 */
void pool_shutdown(void)
{
    for(int i=0; i<NUMBER_OF_BEES; ++i) {
        pthread_cancel(bee[i]);
        pthread_join(bee[i], NULL);
    }
    sem_destroy(&sem);
    pthread_mutex_destroy(&mutex);
}
os@os-VirtualBox:~/Desktop/proj4$
```