운영체제론 Project 4

2017012251 윤영훈

소스코드 설명

※ 미리 스켈레톤으로 구현되어있던 코드는 설명하지 않음.

```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"

/*
 * 스레드 풀의 FIFO 대기열 길이와 일꾼 스레드의 갯수를 지정한다.
 */
#define QUEUE_SIZE 10
#define NUMBER_OF_BEES 3

int head = 0;
int tail = 0;
int worktodo_num = 0;

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

구현을 위해 필요한 변수 head, tail, worktodo_num을 line 18-20을 통해 선언.

head : dequeue() 실행 시 worktodo 대기열에서 가져올 task의 index를 저장.
tail : enqueue() 실행 시 worktodo 대기열에 task를 집어넣을 index를 저장.
worktodo_num : 현재 worktodo 대기열에 존재하는 task의 수.

```c
static int enqueue(task_t t)
{
    pthread_mutex_lock(&mutex);

    if (worktodo_num < 10){
        worktodo[tail] = t;

        if (tail == QUEUE_SIZE-1)
            tail = 0;
        else
            tail++;

        worktodo_num++;

        pthread_mutex_unlock(&mutex);
        return 0;
    }
    else {
        pthread_mutex_unlock(&mutex);
        return 1;
    }
}
```

enqueue() 함수 구현.

worktodo 대기열에 접근할 때 lock을 위해 line 46을 통해 mutex lock을 걸어준다.

이후 worktodo 대기열이 가득 차있지 않다면 대기열에 task_t t를 집어넣고 tail을 변경시키는데, worktodo 대기열은 원형 buffer 이므로 tail이 last index 9를 점유하고 있다면 0으로 만들고, 그렇지 않다면 1 증가시킨다.
이후 worktodo_num을 1 증가시키고 line 58을 통해 mutex lock을 해제하고 0을 return한다.

worktodo 대기열이 가득 찼다면 추가적인 조치 없이 바로 line 62를 통해 mutex lock을 해제하고 1을 return한다.

```
71  static int dequeue(task_t *t)
72  {
73      pthread_mutex_lock(&mutex);
74
75      if (worktodo_num > 0){
76          *t = worktodo[head];
77
78          if (head == QUEUE_SIZE-1)
79              head = 0;
80          else
81              head++;
82
83          worktodo_num--;
84
85          pthread_mutex_unlock(&mutex);
86          return 0;
87      }
88      else {
89          pthread_mutex_unlock(&mutex);
90          return 1;
91      }
92  }
```

dequeue() 함수 구현.

enqueue() 와 동일하게 대기열에 접근할 때 line 73을 통해 mutex lock을 걸어준다.

이후 worktodo 대기열이 전부 비어있지 않다면 parameter로 받은 task_t *t에 worktodo[head]의 내용을 입력한다.
이때 worktodo 대기열은 원형 buffer 이므로 head가 last index 9를 점유하고 있다면 0으로 만들고, 그렇지 않다면 1 증가시킨다. 이후 worktodo_num을 1 감소시키고 line 85을 통해 mutex lock을 해제하고 0을 return한다.

worktodo 대기열이 전부 비어있다면 추가적인 조치 없이 바로 line 89를 통해 mutex lock을 해제하고 1을 return한다.

```
98  static pthread_t bee[NUMBER_OF_BEES];
99  static sem_t sem;
100
101 /*
102  * 풀에 있는 일꾼 스레드로 FIFO 대기열에서 기다리고 있는 작업을 하나씩 꺼내서 실행한다.
103  */
104 static void *worker(void *param)
105 {
106     for(;;){
107         task_t new_task;
108
109         if (dequeue(&new_task) == 0)
110             (*new_task.function)(new_task.data);
111
112         sem_wait(&sem);
113     }
114
115     pthread_exit(0);
116 }
```

bee는 3개의 일꾼 thread의 ID를 저장하는 배열이고, sem은 counting semaphore로 대기열에 입력된 작업의 개수이다.

worker() 함수 구현.

worker는 pool에 존재하는 일꾼 thread이다. 따라서 worktodo 대기열에서 task를 dequeue() 하여 new_task에 저장하고, dequeue() 에 성공했다면 line 110을 통해 불러온 task를 실행한다.

대기열에서 task를 가져왔으므로, line 112의 sem_wait()을 통해 sem의 값을 1 감소시킨다.

이후에 나올 pthread_cancel(bee[i])를 통해 일꾼들이 cancel되고 line 115의 pthread_exit(0)을 통해 이후에 나올 pthread_join() 함수에 종료를 알린다.

```
122  int pool_submit(void (*f)(void *p), void *p)
123  {
124      task_t new_task;
125
126      new_task.function = f;
127      new_task.data = p;
128
129      if (enqueue(new_task) == 0){
130          sem_post(&sem);
131          return 0;
132      }
133      else
134          return 1;
135  }
```

pool_submit() 함수 구현.

task_t new_task를 생성하고, 여기에 parameter로 입력받은 function f와 data p를 채워넣어 new_task를 완성한다.
이를 enqueue()를 통해 대기열에 집어넣는다.

대기열에 task를 집어넣었으므로, line 130의 sem_post()를 통해 sem의 값을 1 증가시키고 0을 return한다.

만약 대기열에 task를 집어넣지 못했으면 1을 return한다.

```
140  void pool_init(void)
141  {
142      pthread_mutex_init(&mutex, NULL);
143
144      for (int i = 0; i < NUMBER_OF_BEES; i++)
145          pthread_create(bee, NULL, worker, NULL);
146
147      sem_init(&sem, 0, 0);
148  }
```

pool_init() 함수 구현.

mutex를 초기화한다.
worker()를 수행하는 3개의 일꾼 thread를 생성한다.
semaphore sem을 초기화한다. 이때 처음에는 아무런 task도 없기에 초기값을 0으로 설정한다.

```c
void pool_shutdown(void)
{
    for(int i = 0; i < NUMBER_OF_BEES; i++)
        pthread_cancel(bee[i]);
    for(int i = 0; i < NUMBER_OF_BEES; i++)
        pthread_join(bee[i], NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&sem);
}
```

pool_init() 함수 구현.

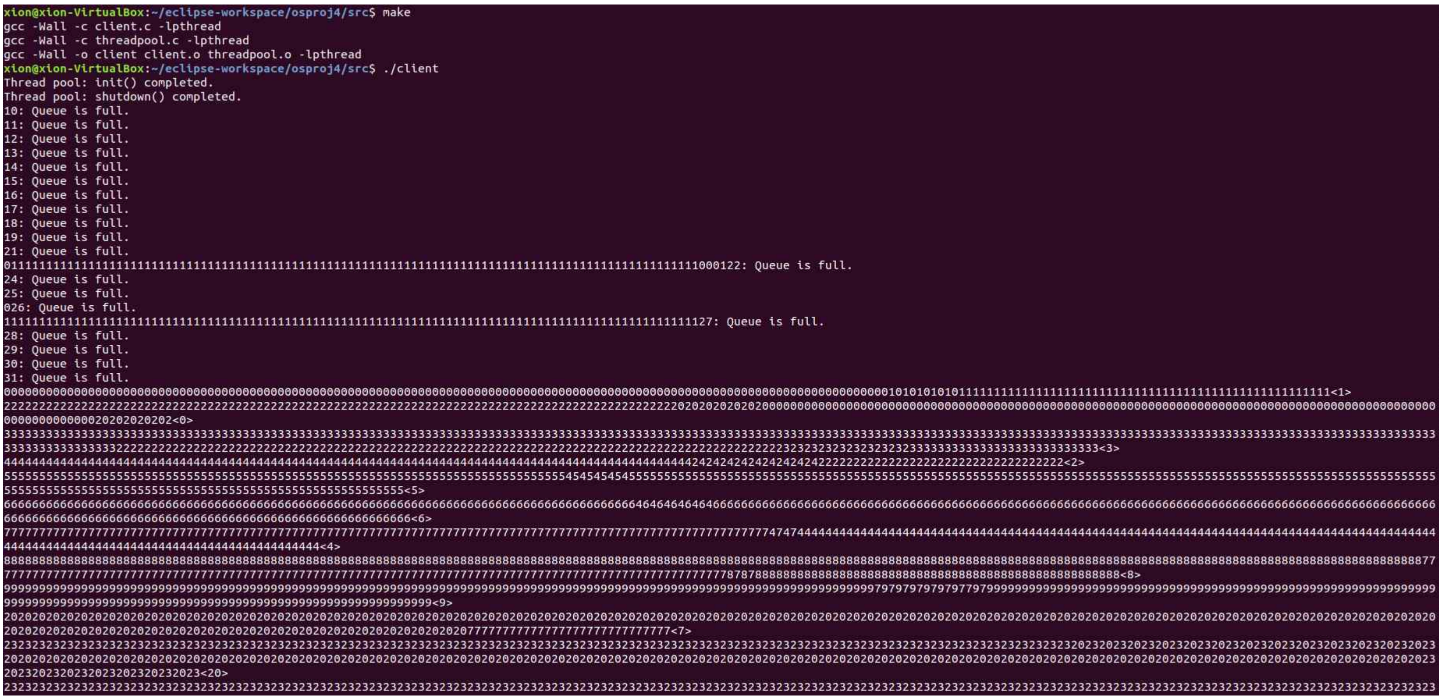
pthread_cancel()을 통해 3개의 일꾼 thread를 모두 cancel하고, 이후 pthread_join()을 통해 각 일꾼 thread들의 pthread_exit(0)를 기다린다.

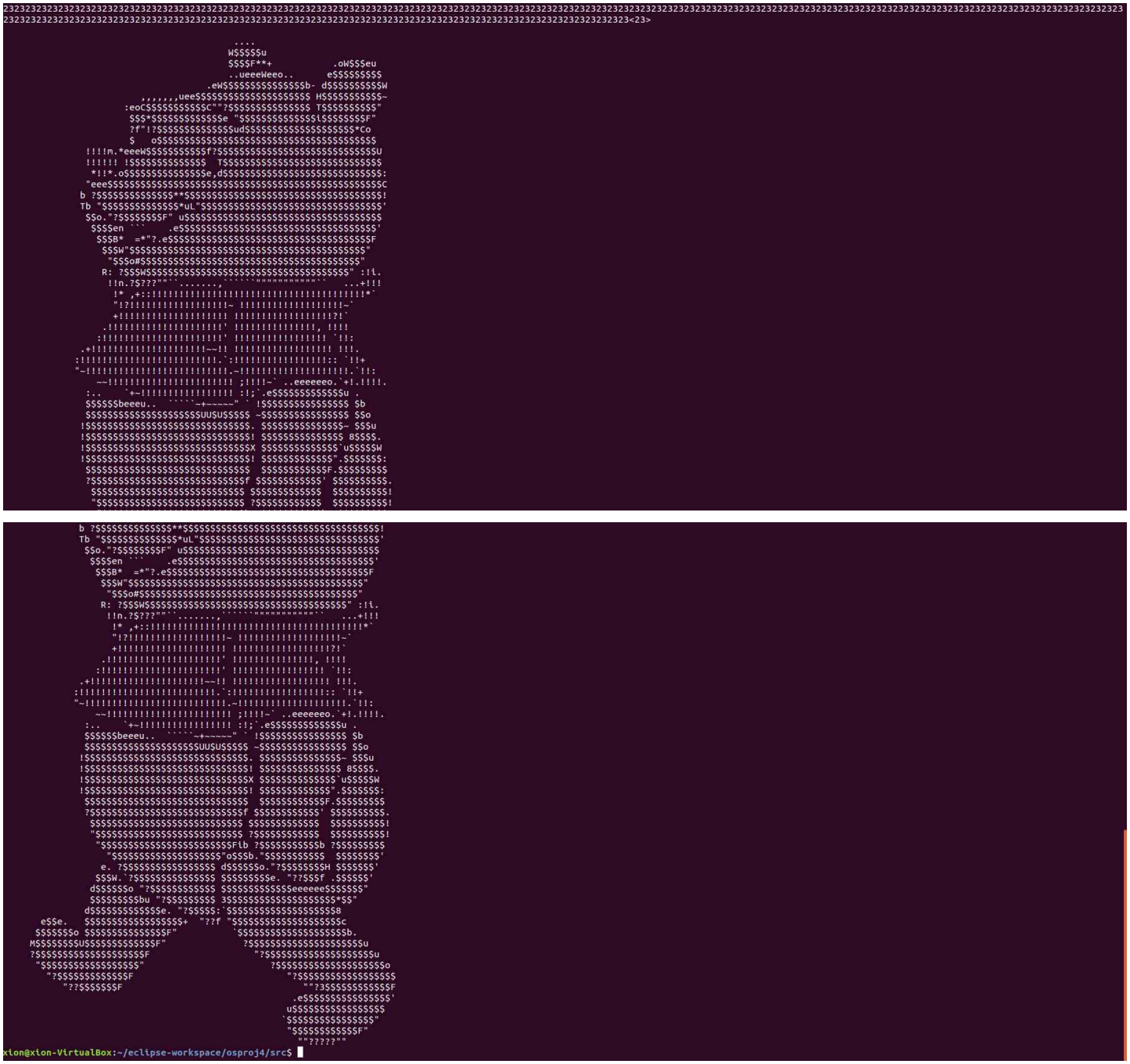
이후 mutex와 sem을 각각 pthread_mutex_destroy, sem_destroy를 통해 삭제하고, 할당된 자원을 해제한다.

컴파일 및 결과물

```
xion@xion-VirtualBox:~/eclipse-workspace/osproj4/src$ make
gcc -Wall -c client.c -lpthread
gcc -Wall -c threadpool.c -lpthread
gcc -Wall -o client client.o threadpool.o -lpthread
xion@xion-VirtualBox:~/eclipse-workspace/osproj4/src$
```

컴파일





Result

10개의 number task가 대기열에 입력되고, 이후에 입력된 task들은 대기열이 가득 차서 요청이 거절되고, 오류 메시지를 출력한다. 이때 중간에 0과 1에 대한 number task가 실행되고, 해당 task들이 대기열에서 빠져나오면서 남은 자리를 20과 23에 대한 number task가 입력되는 것을 볼 수 있다. 이후 31까지의 number task는 모두 거절되고, 대기열에 입력된 number task들이 수행된다. 마지막 23에 대한 number task까지 모두 수행되고 난 뒤 대기열이 빈 상태가 된다.

이후 pool_submit(donothing, NULL); pool_submit(donothing, NULL); pool_submit(pooh, NULL);이 실행되고, 대기열은 빈 상태이고 일꾼 thread는 3개가 있으므로 바로 3개의 task 모두 방해없이 실행된다. 이때 donothing은 아무 것도 하지 않는 함수이므로 곰돌이 푸를 출력하는 pooh 함수만 출력되게 된다.

참고로 해당 result를 수행한 후 터미널을 종료해버려서 프로그램 실행 결과가 담긴 client.txt file의 내용은 위 result 사진의 내용과 다르고, ‘./client > client.txt’를 통해 결과물을 출력한 것이 원인인지 Result 사진처럼 뒤섞이지 않고 일부 상대적으로 정렬된 결과가 출력되었다.