운영체제론 Project #4
스레드풀

작성자 소프트웨어학부
2018045241 홍산해
2020062321 이은비
2020088568 백홍열

목 차

Ⅰ 스레드풀 알고리즘

i. pthread_pool_init

1. 전달받은 bee_size와 queue_size의 유효성을 검사한다. 요청한 bee_size가 최대 일꾼 수보다 크거나, queue_size가 최대 길이보다 크다면 POOL_FAIL을 리턴하며 종료한다. 만약, queue_size(대기열의 길이)가 bee_size(일꾼의 수)보다 작다면 queue_size가 최소한 bee_size가 되도록 상향 조정한다.
2. 일꾼 스레드가 들어갈 스레드풀 배열(bees)과 요청받은 작업을 넣을 대기열 배열(q)를 만든다.
3. Pool의 변수를 초기화한다. Worker를 작동시키기 위해 running을 true로, q_size와 bee_size를 전달받은 queue_size와 bee_size로 초기화시킨다. 앞서 만든 스레드풀 배열 bees와 대기열 배열 q를 pool의 bee, q로 초기화한다.
4. 대기열을 조회, 변경하기 위해 사용할 mutex와 대기열에 작업이 채워지길 기다리는 full 조건변수, 대기열에 빈자리가 생기길 기다리는 empty 조건변수를 만든다.
5. 그 후 worker를 실행하는 bee_size만큼의 일꾼 스레드를 만들고, POOL_SUCCESS를 리턴 뒤 종료한다.

ii. worker

1. 전달받은 pool을 조회하기 위해 mutex 락을 건뒤, pool의 running이 ture이면 다음을 반복한다.
2. 대기열이 비었다면 조건변수 full을 통해 대기열에 작업이 생길 때 까지 대기한다.
3. submit을 통해 작업이 생겼다면, submit은 full signal을 보내고 이에 의해 깨어나게 된다.
4. 만약 shutdown의 broadcast에 의해 깨어났다면, 2번 과정 앞으로 돌아가 running의 상태를재확인한다. running의 상태가 false라면 반복을 빠져나와 mutex 락을 해제하고 스레드를 종료한다.
5. 깨어난 일꾼 스레드는 대기열에서 작업을 꺼내고, 대기열의 길이와 다음 실행될 위치를 수정한다. 해당 일꾼 스레드가 작업 하나를 꺼내 빈자리가 생겼으므로 empty 조건변수에 signal을 보낸다.
6. mutex 락을 해제 후 꺼낸 작업을 수행한다. 그리고 다시 pool을 조회하기 위해 mutex 락을 얻고 2번 과정으로 돌아가 스레드풀이 종료될 때까지 반복한다.

iii. pthread_pool_shutdown

1. 전달받은 pool을 조회하기 위해 mutex 락을 건뒤, 일꾼을 종료시키기 위해 running을 비활성화한다.
2. full, empty 조건변수를 기다리는 스레드들을 broadcast로 모두 깨운 뒤, mutex 락을 해제한다. 그리고 모든 일꾼 스레드의 종료를 기다리고 자원을 반납한다.

iv. pthread_pool_submit

1. 전달받은 pool을 조회하기 위해 mutex 락을 건뒤, 대기열에 있는 작업이 꽉 차있는지를 확인한다. 만약, 꽉 차있고 전달받은 flag가 POOL_NOWAIT이라면 mutex 락을 해제후 POOL_FULL을 리턴 뒤 종료한다. 만약, 꽉 차있고 전달받은 flag가 POOL_WAIT이라면 대기열에 빈자리가 생길 때까지 조건변수 empty를 통해 대기한다.
2. 대기열의 빈자리에 전달받은 작업을 추가하고, 대기열의 길이를 수정한다. 대기열에 작업을 추가하였으므로 full 조건변수에 signal을 보낸다. mutex 락을 해제후 POOL_SUCCESS를 리턴하고 종료한다.

II 컴파일 및 실행

1. 컴파일 과정

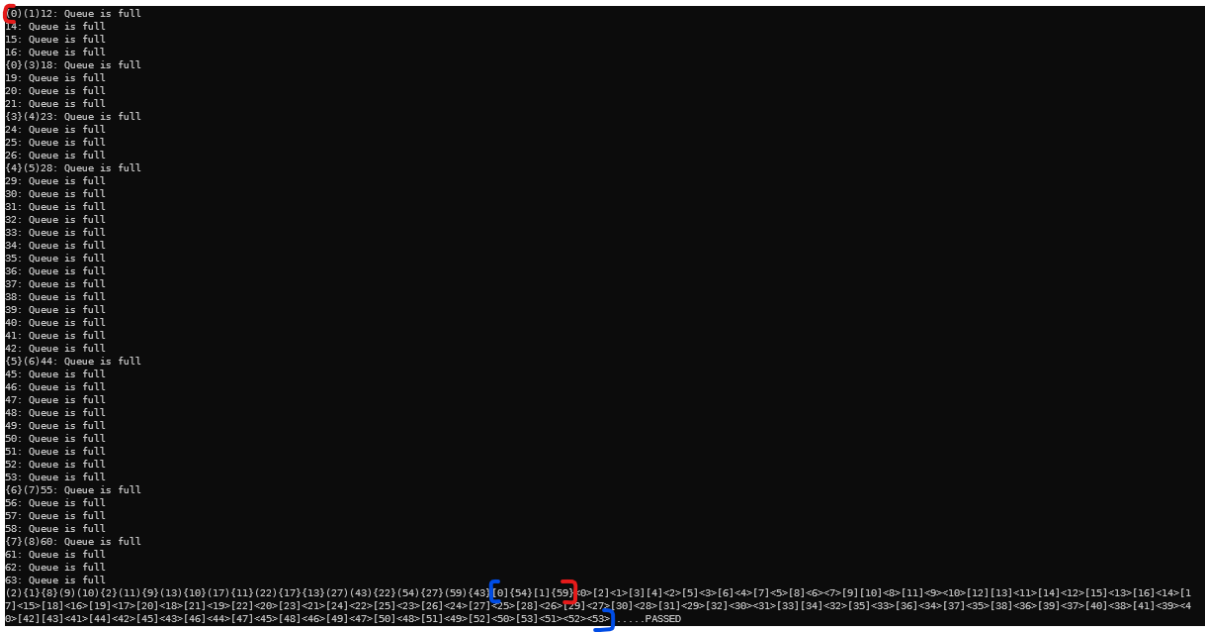
```
 sanhaehong@admin ~/operation_system/proj4 ls
Makefile  client.c  pthread_pool.c  pthread_pool.h
 sanhaehong@admin ~/operation_system/proj4 sudo make
[sudo] password for sanhaehong:
gcc -Wall -c client.c
gcc -Wall -c pthread_pool.c
gcc -Wall -o client client.o pthread_pool.o -lpthread
 sanhaehong@admin ~/operation_system/proj4 ls
Makefile  client  client.c  client.o  pthread_pool.c  pthread_pool.h  pthread_pool.o
 sanhaehong@admin ~/operation_system/proj4
```

주어진 Makefile을 통해 sudo make 명령어를 실행하여 컴파일한다

2. 실행 및 설명

```
 sanhaehong@admin ~/operation_system/proj4 ./client
pthreaad_pool_init(): 일꾼 스레드 최대 수 초과.....PASSED
pthreaad_pool_init(): 대기열 최대 용량 초과.....PASSED
pthread_pool_init(): 완료......PASSED
pthread_pool_shutdown(): 완료......PASSED
```

처음에는 client.c에 따라 queue_size와 bee_size의 크기에 대한 예외처리 유무를 확인한다.
이후에는 init 이후 shutdown이 제대로 작동하는지를 확인한다
확인이 완료되면 PASSED를 출력하므로, 프로그램이 정상적으로 작동함을 확인했다.



이후에는 일꾼 스레드가 3개이고 대기열의 크기가 10개인 스레드풀 pool1을 가동한뒤,
플래그를 POOL_NOWAIT로 하여 NTASK(64)개의 number1 업무를 등록한다.
(number 1은 어떤 정수 x를 인자로 받아서 (x){x}를 화면에 출력한다.)

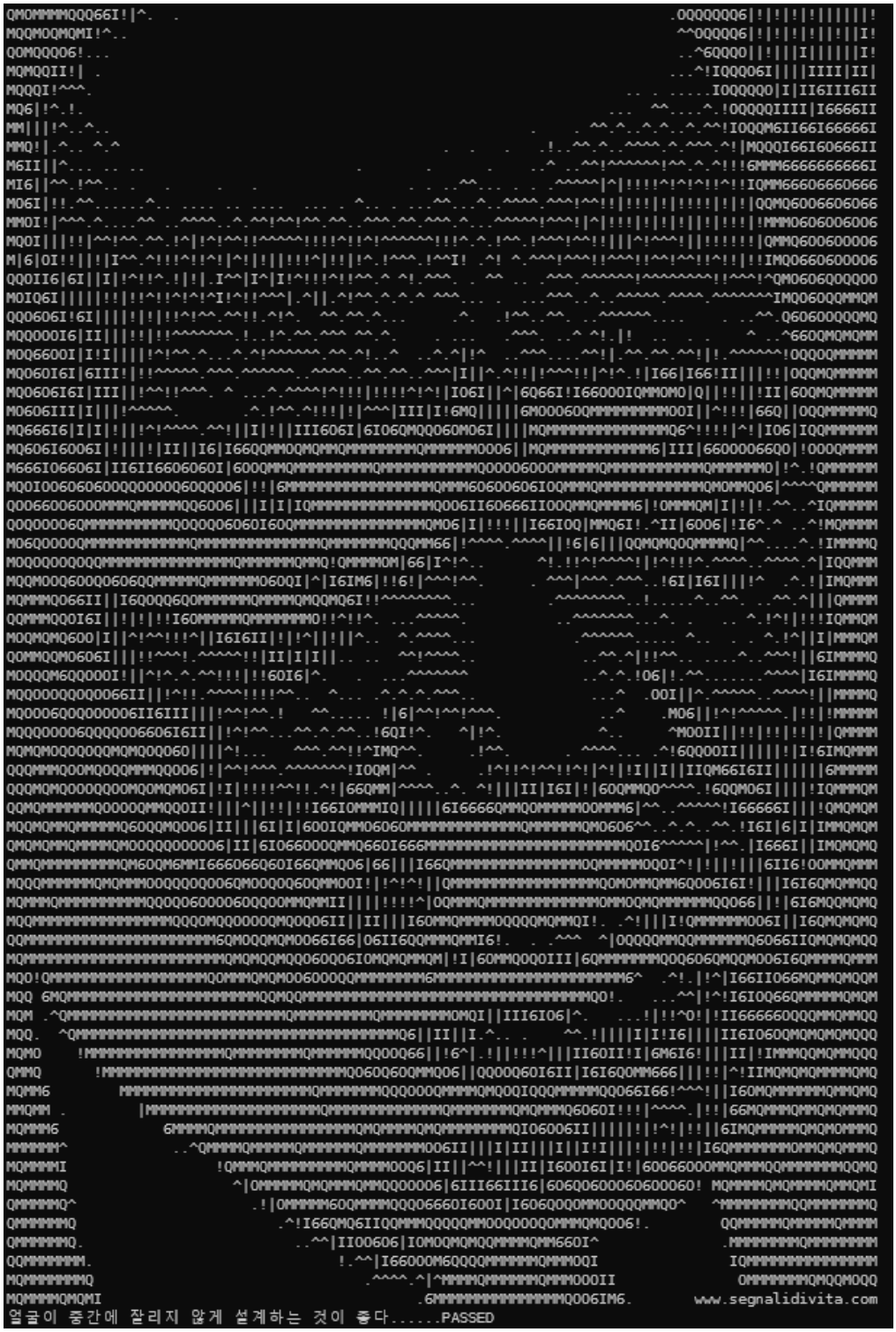
대기열의 크기에 비해 업무가 매우 많으므로 대기열이 가득 차는 현상이 빈번하게 일어난다.
이 때 플래그가 POOL_NOWAIT이므로,
작업 요청에 실패하여 Queue가 꽉 찼다는 메시지를 여러 번 출력한다.
또한 number1은 (x)와{x}사이에서 잠깐 쉬기 때문에 불규칙적으로 출력한다.

이후에는 플래그를 POOL_WAIT로 한뒤 NTASK(64)개의 number2 업무를 등록한다.
(number 2는 어떤 정수 x를 인자로 받아서 [x]<x>를 화면에 출력한다.)

마찬가지로 number2은 [x]와<x>사이에서 잠깐 쉬기 때문에 불규칙적으로 출력한다. 또한 대기 시간동안 number1의 출력이 number2를 앞질러 두 출력의 순서가 뒤바뀔수 있다.

number2도 1과 마찬가지로 대기열이 가득 차는 현상이 빈번하게 일어난다.
그런데 number2 업무의 경우 플래그가 POOL_WAIT이므로,
대기열이 꽉 찼을 때 condition 변수 empty를 통해 대기하기 때문에 작업은 항상 성공한다.

마지막으로 shutdown을 통해 모든 스레드를 종료했으므로 <53>에서 출력을 멈춤을 확인했다.

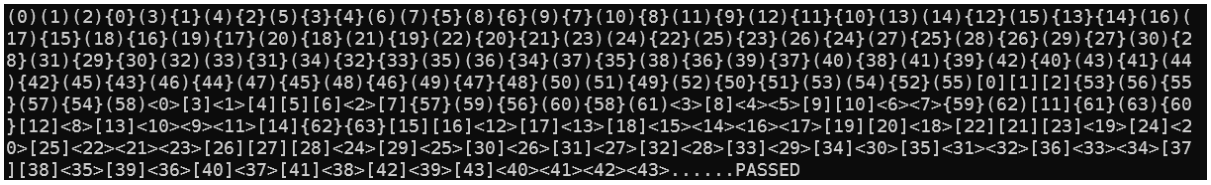


다음으로 일꾼 스레드가 8개이고 대기열의 크기가 16개인 스레드풀 pool1을 가동한뒤,

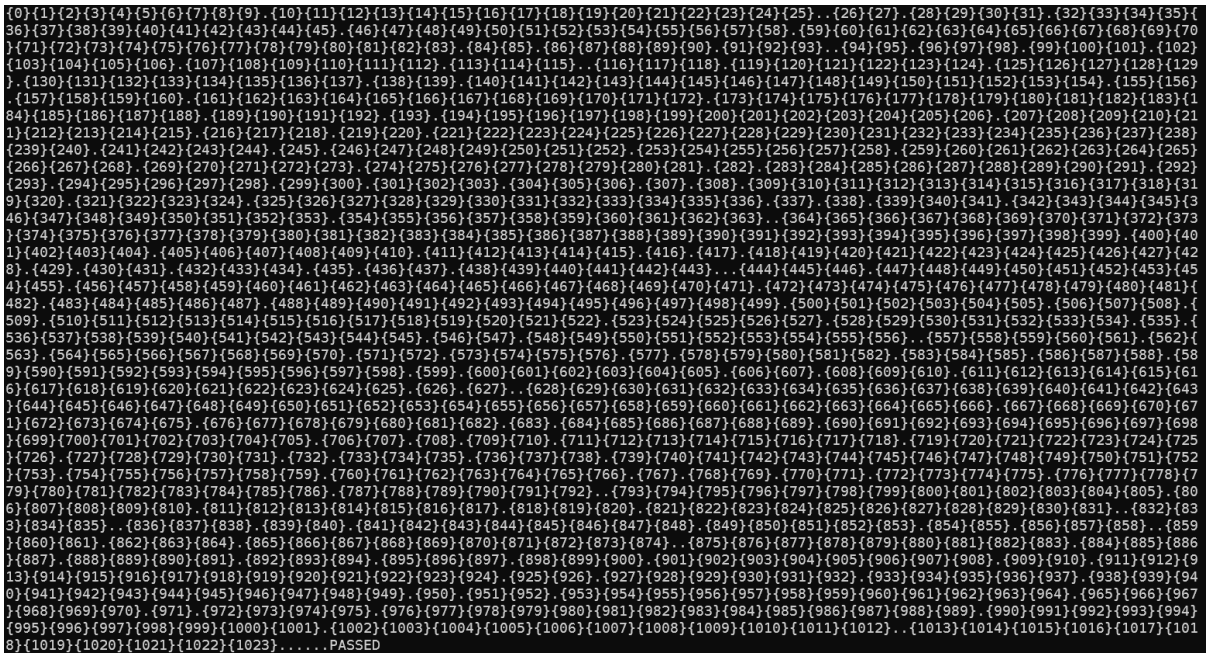
플래그를 POOL_WAIT로 하여 face(업무)를 등록한다.

(face는 ASCII 문자로 된 어떤 얼굴을 출력하는 함수이다.)

number1, number2와는 달리 얼굴을 출력하는 업무는 1개의 face 뿐이므로, 중간에 출력이 종료되지 않고 끝까지 출력한뒤 자연스럽게 종료됨을 확인했다.



다음으로는 일꾼 스레드가 3개이고 대기열의 크기가 10개인 스레드풀 pool1과
일꾼 스레드가 5개이고 대기열의 크기가 20개인 스레드풀 pool2를 가동한다.
그리고 각각의 스레드풀에 NTASK(64)개의 number1, number2 업무를 등록한다.
앞에서 나왔던 출력과 유사하며, 마찬가지로 마지막에 shutdown을 통해 스레드풀 pool2를 종료했으므로 <43>에서 출력을 멈춘 것을 확인했다.



마지막으로 스레드풀 pool2, 3, 4를 크기와 대기열을 무작위로 하여 가동한다.
그리고 각각의 스레드풀에 NLOOP(1024)개의 dot 업무를 등록한다.
(dot는 1/16384 확률로 점을 출력한다.)
남아있는 스레드풀 pool1에 i(for문의 변수)를 인자로 하는 number3 업무를 등록한다.
(number3는 어떤 정수 x를 인자로 받아서 화면에 출력한다.)
이후에는 shutdown을 통해 스레드풀 pool2, 3, 4를 종료한다.

그리고 이 과정을 NLOOP(1024)번 반복한다.

Number1는 NLOOP(1024)개의 업무가 등록되어 0 ~ 1023을 출력하고,
각각의 스레드풀 pool2, 3, 4는 무작위 확률로 중간중간 점을 출력함을 확인했다.

III 느낀점

2020088568 백홍열

destroy 함수를 구현하고 테스트를 하던 중 mutex_lock과 free에서 오류가 발생했습니다.

mutex_lock에서는 Assertion error를, free에서는 segmentation error를 확인할 수 있었습니다.

전자의 오류는 mutex가 초기화 되지 않으면 생기는 오류임을 검색하여 알았지만 init에서 분명히 초기화를 해줬기 때문에 믿지 못하고 다른 발생 사례를 찾아보려 했습니다.

하지만 그 외의 사례는 찾을 수 없어서 init에서의 변수들의 주소값과 destroy에서의 주소값을 비교해보았습니다.

주소값이 다르게 나와 이유를 찾아보니 main에서 주소값을 넘겨준 pool 포인터에 동적할당을 하는 실수를 했음이 밝혀졌습니다.

main의 pool1변수가 포인터라고 착각하고 코드를 짜서 생긴 문제였습니다.

구현할 함수로 들어오는 값이 무엇인지 확실하게 확인해야 했음을 깨닫게 되었고, pool1이 포인터였다고 하면 init 함수의 return값이 할당한 주소값이어야 했음을 뒤늦게 생각해낼 수 있었습니다.

포인터에 대한 이해가 더 필요함을 느끼게 되었습니다.

처음 worker함수를 구현할 때 pool->running 이 참인 동안에 실행하도록 하면서 이 변수 값은 lock없이 읽어올 수 있지 않을까 생각하였습니다.

이 변수는 처음 init에 의해 true로 세팅된 후 destroy에서만 변경되기 때문에 reader들은 동시 접근을 해도 괜찮다고 보고 코드를 만들었습니다.

하지만 데드락이 걸리는 것을 경험하고 이 데드락의 발생이유를 찾으려 했습니다.

destroy함수에서는 running변수를 먼저 false로 한 후 broadcast해서 스레드들을 깨웁니다. 스레드들이 깨어나기 전에 running이 false로 바뀌기 때문에 문제가 없을 것이라고 생각했지만 데드락은 발생했기 때문에 고민를 거듭한 끝에 시나리오 하나를 생각해낼 수 있었습니다.

destroy가 호출 되었을 때 task를 진행하던 스레드가 종료 후 destroy보다 먼저 running에 접근하고 broadcast 이후에 cond_wait을 하게 된다면 데드락이 생길 수 있었습니다.

데드락을 통해 상호배타에 대해서는 조금 더 신중했어야 했음을 깨달았고 worker가 running에 lock을 가지고 접근하게 한 후 정상 작동함을 확인할 수 있었습니다.

2020062321 이은비

worker와 submit에서 대기열이 채워지거나 빈자리가 생길 때까지의 상황을 일꾼 스레드가 대기열을 읽으며 계속 체크한다면 자원이 낭비되는 문제가 있다.

따라서 조건변수의 사용을 통해 대기열이 채워지고, 빈자리가 생기는 signal을 주고받아 해결할 수 있다는 것을 알게 되었다. 또한 스레드가 락을 소유한 채로 종료된다면 데드락이 발생됨을 주의하며 프로그램을 작성해야 하는 것을 알게 되었다.

2018045241 홍산해

스레드풀 과제를 통해 condition, mutex를 통한 다중 스레드 관리 방법에 대해 좀 더 심도있는 공부를 할 수 있었다.

과제 초반 pool 구조체의 초기화 단계에서 구조체 포인터를 수정으로 인한 에러가 지속적으로 발생했다. 포인터 개념이 미비함을 깨닫고 복습에 대한 필요성을 느꼈다.

또한 worker에서 pool에 대한 변수를 조회할 때 데드락의 가능성을 배제한채로 코드를 짜 디버깅하는데 시간을 많이 소비했다.

다음부터는 코드를 짤 때 좀 더 차근차근 생각해보고 코드를 짜야 겠다는 생각이 들었다.