과제 #4 : Ptheread

○ 과제 목표

- 다중 쓰레드 프로그래밍의 기본 개념을 이해
- lock 및 조건 변수를 사용하는 쓰레드(pthread) 프로그램 구현

○ 기본 지식

- pthread 사용자 수준 쓰레드 라이브러리
  - ✓ https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
  - ✓ https://computing.llnl.gov/tutorials/pthreads/
- 다중 쓰레드로 구현된 시스템의 일반적인 형태
  - ✓ 서버에는 하나 이상의 마스터 쓰레드와 워커 쓰레드 풀 존재
  - ✓ 웹 클라이언트에서 새로운 연결이 도착하면 마스터 쓰레드는 이 요청을 수락하고 이를 워커 쓰레드 중 하나에게 이 요청을 넘김
  - ✓ 워커 쓰레드는 네트워크 소켓에서 웹 요청을 읽고 클라이언트에 다시 응답

○ 과제 기본 내용

- 많은 응용에서 볼 수 있는 간단한 마스터 및 워커 스레드 풀을 구현
  - ✓ 마스터 쓰레드는 연속적으로 숫자를 생산하고 이를 버퍼에 둠
- 읽기-쓰기 Locks을 통한 쓰레드 간 병행성 향상 프로그램 구현
- pthread를 이용한 사용자 수준의 SSU_세마포어 구현

○ 과제 구현 내용

- 1. Master-Worker Thread Pool 구현
  - ✓ 간단한 마스터-워커 쓰레드 풀 구현
  - ✓ 마스터 쓰레드는 연속적으로 숫자를 생성하여 버퍼에 넣고 워커 쓰레드는 이를 소비 (표준 출력)
  - ✓ 주어진 코드인 master-worker.c 코드 추가 (코드 완성)
    - % gcc master-worker.c -lpthread (pthread 라이브러리 링크)
    - 4개의 명령어 인자
      - ☞ (1) 생성할 숫자 (M)
      - ☞ (2) 생성 된 숫자(정수형만) 를 저장할 버퍼 최대 크기 (N)
      - ☞ (3) 생성한 숫자(정수형만)를 소비하는 워커 쓰레드 수 (C)
      - ☞ (4) 숫자를 생성 할 마스터 쓰레드의 수 (P)
    - 주어진 코드 코드는 0에서 M-1까지의 생성할 숫자(정수형)를 공유 버퍼 배열로 생성하는 P개의 마스터 쓰레드 생성하고, 생성된 마스터 쓰레드들이 join될 때까지 기다린 다음 종료함.
      - ☞ 주어진 코드인 master-worker.c 에는 동기화가 포함되어 있지 않음을 유의.
  - ✓ master-worker.c 코드 추가(완성) 방법
    - 필요한 수(C) 만큼의 워커 쓰레드를 생성하는 코드를 추가하고 워커 쓰레드가 실행할 함수 작성
      - ☞ 워커 쓰레드가 실행할 함수는 공유 버퍼에서 항목을 제거하고 소비(표준 출력).
    - 모든 숫자가 정확히 한 번 생성되고 한번 소비되도록 생산자(마스터 쓰레드)와 소비자(워커 쓰레

드)를 올바르게 동기화하는 코드 추가

- 생산자(마스터 쓰레드)는 버퍼가 꽉 찼을 때 더 이상 생산을 시도해서는 안 되며, 소비자(워커 쓰레드)는 빈 버퍼에서 소비하면 안됨
- C개의 모든 워커 쓰레드가 숫자 소비에 관여해야겠지만, 워커 쓰레드 간 완벽한 로드밸런싱(부하분산, C개의 워커 쓰레드가 소비하는 양이 거의 동일하게)보장 할 필요 없음
- 모든 M개의 숫자 (0에서 M-1까지)가 생성되고 소비되면, 모든 쓰레드는 종료
- main() 쓰레드는 마스터 쓰레드 및 워커 쓰레드에서 pthread_jon()을 호출하고 모든 쓰레드가 join 되면 스스로 종료
- waiting과 signaling을 위해 pthreads에서 제공하는 조건 변수만 사용
- busy waiting은 허용하지 않음

✓ 코드가 올바르게 작성되면 0에서 M-1까지의 모든 숫자가 마스터 쓰레드에 의해 정확히 한 번 생산되고 워크 쓰레드에 의해 정확히 한 번 소비됨

✓ 구현 테스트

- 주어진 check.awk 이용
  - ☞ awk 파일 참고 문서 : https://recipes4dev.tistory.com/171 와 https://systemdesigner.tistory.com/45
  - ☞ check.awk 스크립트는 마스터 쓰레드와 워커 스레드가 호출해야하는 두 가지 print() 의존
  - ☞ 마스터 쓰레드는 버퍼에 숫자를 넣을 때 생성되는 숫자를, print() 사용하여 표준 출력
  - ☞ 작업자 쓰레드는 버퍼에서 숫자를 제거할 때 소비되는 숫자를, print() 사용하여 표준 출력
  - ☞ check.awk 스크립트에서 출력을 parsing(구문분석)하기 때문에 print() 수정하면 안됨

- 2. Reader-Writer Locks 쓰레드 프로그램 구현
  - r_lock()과 w_lock()으로 공유 데이터를 처리하여 쓰레드 간 병행성 향상

✓ Reader(읽기)이던 Writer(쓰기)이던 쓰레드들이 동시에 데이터를 접근하면 안됨

✓ 두 가지 유형의 읽기/쓰기 lock 구현

- ☞ (시나리오) 읽기 쓰레드 R1이 읽기를 위해 read/write lock을 모두 획득했다고 가정한 상태에서 쓰기 쓰레드인 W와 다른 읽기 쓰레드인 R2가 모두 lock을 요청하는 상태라고 가정할 때
- ☞ 읽기만 하는 R2도 R1과 동시에 잠금을 획득하도록 허용하는 것이 더 좋을 수 있으나, R2에게 lock을 허용하게 되면 W는 R1와 R2 모두 lock을 해제 할 때까지 기다려야하고 이로 인해 W의 대기 시간이 길어질 수 있음.
- ☞ 쓰기 쓰레드가 대기 중일 때 더 많은 읽기 쓰레드에게 lock을 허용하는 것은 설계하는 사람의 판단에 따름

- (첫 번째 방법) read/write lock을 읽기 쓰레드의 기본 설정으로 구현 => 쓰기 쓰레드가 lock을 획득하기 위해 기다리고 있는 경우, 추가적으로 읽기 쓰레드는 이전 읽기 쓰레드와 동시에 read/write lock을 획득 가능하게 하는 방법
- (두 번째 방법) read/write lock을 쓰기 쓰레드의 기본 설정으로 구현 => 쓰기 쓰레드가 lock 획득을 위해 대기 중일 때는 읽기 쓰레드는 lock을 획득할 수 없음. 즉, 쓰기 쓰레드의 대기 시간은 필요 이상으로 길어지지 않게 하는 방법

✓ 주어진 rw_lock.h에서 read/write lock의 구조를 정의하고 구현

✓ 주어진 rw_lock-r-test.c와 rw_lock-w-test.c에 각각 읽기/쓰기 쓰레드의 기본 설정으로 read/write lock 코드 작성

- init_rwlock() : lock 초기화
- r_lock() : read-only critical section(임계영역)에 들어가기 전에 읽기 쓰레드에 의해 호출

- r_unlock() : read-only critical section에서 나올(종료)할 때 읽기 쓰레드에 의해 호출
- w_lock(): 공유 데이터를 갱신하기 위해 critical section에 들어가기 전에 쓰기 쓰레드에 의해 호출
- w_unlock() : critical section에서 나올(종료)할 때 쓰기 쓰레드에 의해 호출

✓ 구현 테스트
- 주어진 2개의 테스트 프로그램(reader_test.c 및 writer_test.c)과 한 개의 sh 프로그램(rw_lock_test.sh) 이용하여 구현한 프로그램의 기능 검증
  ☞ reader_test.c와 writer_test.c : 다양한 read/write lock을 테스트
  ☞ rw_lock_test.sh : read/write lock을 한번에 수행시킬 수 있는 스크립트
  ☞ 테스트 프로그램은 2개의 명령어 라인 인자(R과 W)를 받고 R 쓰레드, W 쓰레드, 추가 R 쓰레드 를 차례로 생성하고, 생성된 쓰레들은 동일한 read/write lock을 획득하고 긴 시간 동안 lock을 보유하고 있다가, 마지막으로 lock을 해제
- 구현한 프로그램의 정확성 검증
  ☞ lock의 획득 및 해제의 상대적 순서 확인
  ☞ read/write lock을 사용하는 테스트 프로그램을 학생이 작성 (R과 W의 다른 값으로 테스트 프로그램을 호출하면 read/write lock 코드를 테스트 가능)

- 3. pthread를 이용한 사용자 수준 세마포어(SSU_Sem) 구현

✓ Linux 커널에서 제공하는 세마포어와 충돌되지 않는 사용자 수준 세마포어(SSU_Sem) 구현
- SSU_Sem의 기능은 리눅스의 세마포어(semaphore)의 기능과 유사

✓ 주어진 SSU_Sem.h 파일에서 SSU_Sem 구조를 정의

✓ 주어진 SSU_Sem.c 파일에서 이 구조에서 작동하는 SSU_Sem_init(), SSU_Sem_up() 및 SSU_Sem_down() 구현
- SSU_Sem_init() : 지정된 SSU_Sem를 지정된 값으로 초기화
- SSU_Sem_up() : SSSU_Sem의 카운터 값을 1 씩 증가시키고, 대기 중인 쓰레드를 깨움
- SSU_Sem_down() : SSU_Sem의 카운터 값을 1 씩 감소시키고, 값이 음수이면 쓰레드가 블록(차단)되고 문맥이 교환되어 나중에 해당 context가 전환되어 나중에 SSU_Sem_up()으로 깨어남.

✓ 구현 테스트
- SSU_Sem_test.c 프로그램을 이용
  ☞ 두 개의 쓰레드를 생성하고 총 세 개의 쓰레드(main() 포함)가 SSU_Semaphore 공용 변수를 공유.

✓ SSU_Sem의 로직을 구현하기 전에는 새로 생성된 쓰레드는 main() 쓰레드 전에 터미널로 출력

✓ SSU_Sem의 로직을 구현 한 후에는 SSU_Sem로 인해 동기화가 제대로 되어 main() 쓰레드가 먼저 출력
  ☞ 테스트 프로그램을 수정해서는 안되며, SSU_Sem 구현을 테스트하기 위해서만 사용
- SSU_Sem_toggle_test.c 프로그램을 이용
  ☞ 세 개의 쓰레드가 순서에 상관 없이 출력되는 것을 thread0, thread1, thread2, thread0, thread1, thread2 ... 와 같이 순서대로 신행되어야 함
  ☞ 쓰레드 간의 동기화를 위해서 구현한 SSU_Sem를 사용해야 하며, 파일에서 pthread 라이브러리의 mutext나 condition 변수를 직접 사용하면 안됨
- SSU_Sem_test.sh 스크립트를 이용
  ☞ 위 테스트 프로그램을 컴파일하고 실행하여 테스트

○ 과제 제출 내용 및 마감

- 보고서
  - ✓ 개요 또는 구현 내용 (수정한 부분 간단하게 설명)과 실행 결과 캡쳐 등 자율적으로 보고서 작성
  - ✓ 단, 수정한 부분이 있는 소스코드 전체를 보고서에 넣지 말 것
- 제출할 tgz 내 위 세부 과제 1, 2, 3의 구현 결과물을 넣을 thread1, thread2, thread3 등 3개의 디렉토리 생성 (3개의 디렉토리 이름은 각각 thread1, thread2, thread3 이어야 함)
  - ✓ 제출할 tgz에는 보고서 파일과 위 3개의 디렉토리가 존재해야 함.
  - ✓ thread1 디렉토리에는 주어진 파일 중 수정한 master-worker.c 등이 존재 (실행파일 및 수정하거나 추가하지 않은 파일은 넣지 말 것)
  - ✓ thread2 디렉토리에는 주어진 파일 중 수정한 rw_lock.h, rw_lock-r-test.c, rw_lock-w-test.c 등이 존재 (실행파일 및 수정하거나 추가하지 않은 파일은 넣지 말 것)
  - ✓ thread3 디렉토리에 주어진 파일 중 수정한 SSU_Sem.h, SSU_Sem.c 등등이 존재 (실행파일 및 수정하거나 추가하지 않은 파일은 넣지 말 것)
- 2020년 11월 4일 (화) 23시 59분 59초까지 구글클래스룸으로 제출

○ 필수 구현

- 1

○ 배점 기준

- 1 : 10점
- 2 : 40점
- 3 : 50점