**2017029589 컴퓨터소프트웨어학부 류지범**

**운영 체제 HW#4**

**제출 일자 : 2020/04/01**

1. **과제 A**
2. **프로그램 설명**

우선 원래의 프로그램인 multithread\_practice는 1부터 19까지의 값을 각각 25000000번씩 더한 것을 합한 것으로 결과는 4750000000이다. 이때 ThreadFunc 함수를 순차적으로 불러서 실행한다. 이는 1을 25000000번 더한 후 2를 25000000번 더하는 과정을 19까지 하는 것인데, 순차적으로 진행이 되기 때문에 상대적으로 속도가 느리다. 스레드를 사용하면 더하는 과정을 여러 스레드가 나누어서 동시에 처리할 수 있기 때문에 속도 개선에 도움이 된다.

프로그램 구조와 함수에 대해 설명해보겠다. 기본적인 코드의 구조는 multithread\_practice와 동일하다. Multithread\_practice에서는 반복문을 통해 ThreadFunc 함수를 20번 실행했는데, 우리는 이 함수를 스레드 생성을 통해 실행할 것이다. 스레드 생성 함수로는 int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg); 가 있다. 첫번째 인자로 스레드 식별자인 threadID를 넣었고, 두번째 인자는 스레드의 특성이므로 NULL로 하고, 세번째 인자로 우리가 실행할 ThreadFunc 함수를 넣었고, 네번째 인자로는 스레드 함수에 매개변수로 넘길 argument[]를 넣었다. 반복문을 통해서 스레드는 20개가 형성되고 메인 스레드는 다른 스레드들을 기다리고 있다는 메시지를 출력한다. 각각의 스레드는 스케줄링에 따라 일을 동시에 처리한다. 하지만 문제가 하나 있다. 원래 multithread\_practice에선 전역 변수인 result의 값에 number를 직접적으로 더했는데, 순차적으로 진행됐기 때문에 아무 문제가 없었다. 하지만 스레드를 통해 더할 땐 동시적으로 일어나기 때문에 값의 신뢰성이 떨어지게 된다. 그래서 ThreadFunc 내부에서 temp라는 local 변수를 생성해서 이 곳에 더하는 과정을 반복하고 더하는 과정이 끝나면 최종적으로 result에 temp를 더하여 값을 업데이트 해주도록 한다. 마지막으로 메인 스레드는 나머지 스레드들이 다 종료될 때까지 기다려야 하는데 이는 pthread\_join(pthread\_t th, void \*\*thread\_return); 함수를 이용한다. 반복문으로 20개의 스레드를 기다리도록 한다. 스레드는 join이 되는 순간 모든 자원을 반납한다. 마지막으로 result 값을 텍스트이(가) 표시된 사진

자동 생성된 설명출력하고 프로그램을 종료한다.

컴파일은 위와 같은 Makefile을 통해 진행했고, -lpthread 옵션을 추가했다.

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명**Time 결과값 설명**

Real time은 실제 소요 시간이고, user time은 user mode에서의 소요시간, sys time은 kernel mode에서의 소요시간이다. User 영역에서 하는 작업이 더 많기 때문에 user time의 값이 sys time 보다 상대적으로 크다. Real time은 user + sys time보다 큰데 이것은 I/O작업이나 다른 작업을 하는데 소요되는 시간도 포함하기 때문이다.

멀티쓰레드에서는 user time과 system time을 쓰레드 개수를 곱한 값으로 출력해주기 때문에 멀티쓰레드를 사용하지 않은 경우랑 비슷한 time이 나왔다.

1. **시간 차이가 나는 이유 설명**

멀티쓰레드를 이용하지 않을 경우 프로그램에서 더하는 과정을 25000000 \* 20번을 순차적으로 진행해야하지만, 멀티쓰레드를 이용할 경우 스케줄링에 따라 이 과정들을 동시에 나누어서 진행하기 때문에 동일 시간에 더하기하는 회수가 훨씬 많으며 결과적으로 프로그램이 더 빨리 끝날 수 있게 된다.

1. **과제 B**
2. **Mutex 예비레포트**

Mutex는 pthread에서 제공하는 동기화 메커니즘으로 shared resource space에 대한 접근 시간 제어로 동기화를 달성한다. 동기화 메커니즘의 핵심은 상호배제이다.

상호 배제는 잠금 형식으로 이루어진다.

* Atomicity - mutex 잠금(lock)는 atomic operation으로 작동한다. 이는 하나의 쓰레드가 mutex 를 이용해서 잠금을 시도하는 도중에 다른 쓰레드가 mutex 잠금을 할 수 없도록 해준다는 뜻이다. 한번에 하나의 mutex 잠금을 하도록 보증해준다.
* Singularity - 만약 스레드가 mutex 잠금을 했다면, 잠금을 한 쓰레드가 mutex 잠금을 해제 하기 전까지 다른 어떠한 쓰레드도 mutex 잠금을 할수 없도록 보증해준다.
* Non-Busy Wait - 바쁜대기 상태에 놓이지 않는다는 뜻으로, 하나의 쓰레드가 mutex 잠금을 시도하는데 이미 다른 쓰레드가 mutex 잠금을 사용하고 있다면 이 쓰레드는 다른 쓰레드가 lock을 해제하기 전까지 해당 지점에 머물러 있으며 이 동안은 어떠한 CPU 자원도 소비하지 않는다.

Mutex의 생성은 int pthread\_mutex\_init(pthread\_mutex\_t\* mutex, const pthread\_mutex\_attr \*attr);로 한다.

Mutex 잠금을 위해선 int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);을 사용

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex); 을 사용하면 잠금을 얻을 수 없을 겨웅 해당 코드에서 블럭되지 않고 바로 에러 코드를 돌려준다.

Mutex 잠금을 얻은 후 해당 영역에서의 작업을 마친 후 잠금을 해제하기 위해선 int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);를 사용한다.