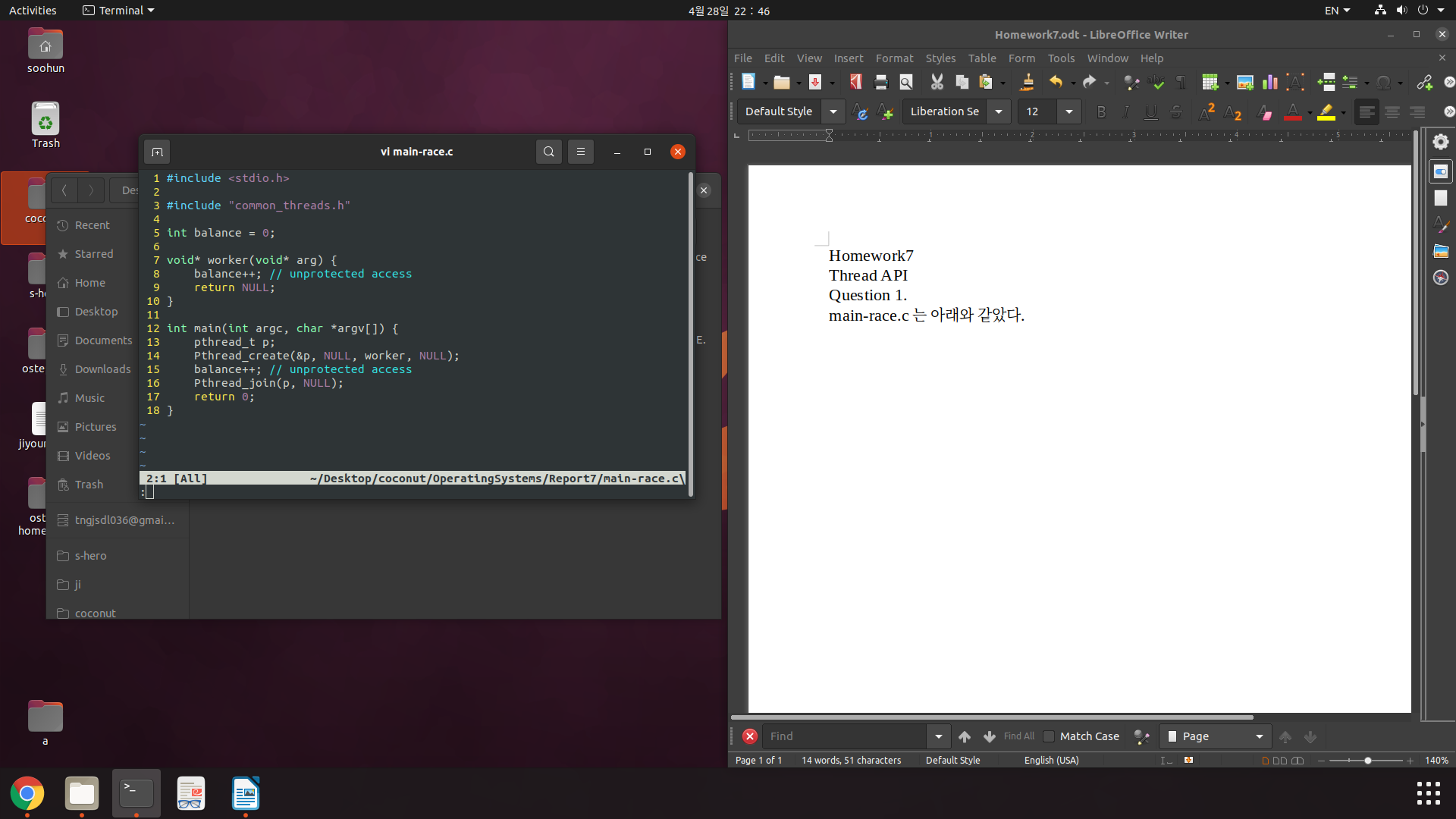
Homework7

Thread API

2016314364 박수헌

Question 1.

main-race.c는 아래와 같았다.

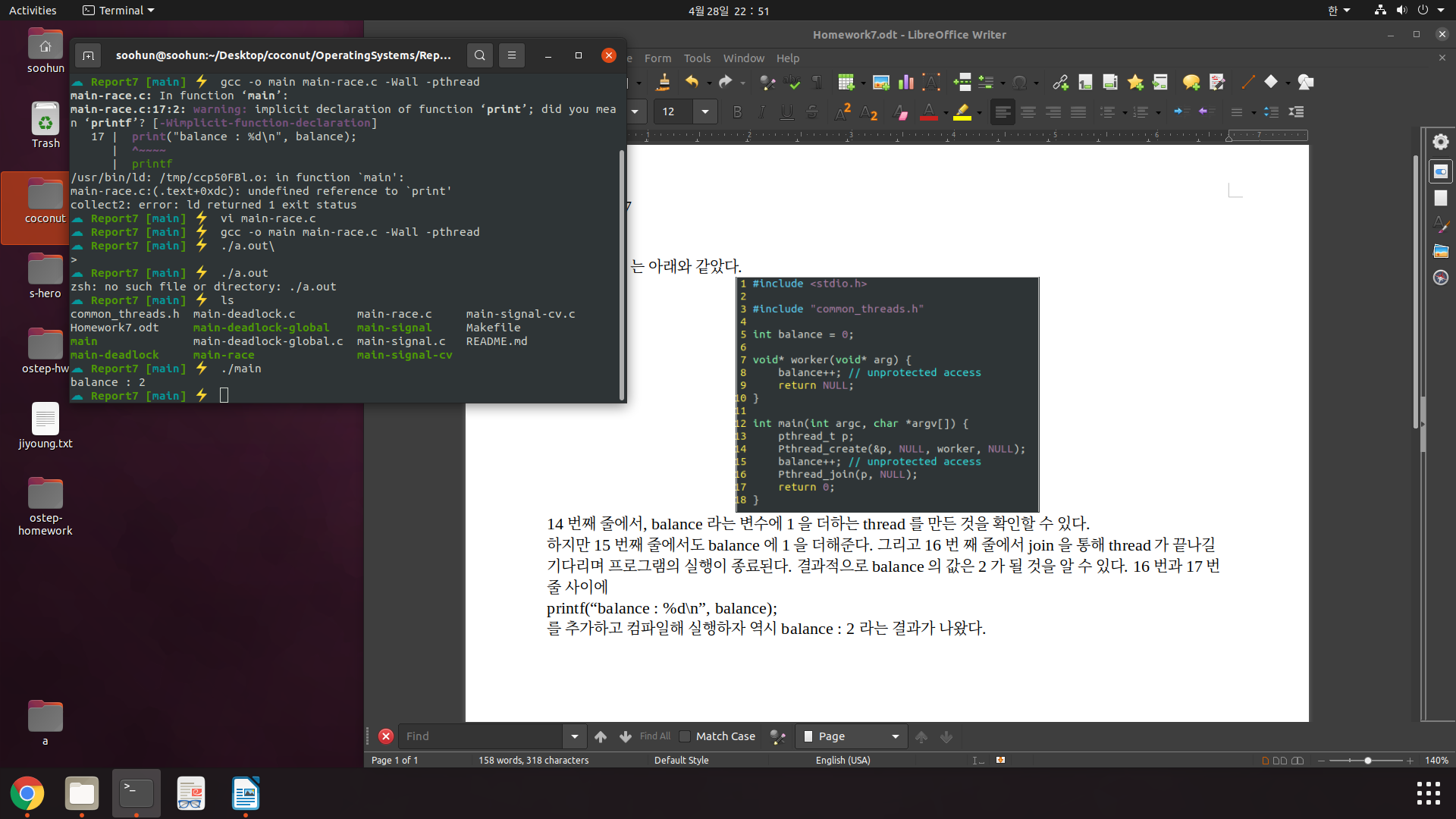


14번째 줄에서, balance라는 변수에 1을 더하는 thread를 만든 것을 확인할 수 있다.

하지만 15번째 줄에서도 balance에 1을 더해준다. 그리고 16번 째 줄에서 join을 통해 thread가 끝나길 기다리며 프로그램의 실행이 종료된다. 결과적으로 balance의 값은 2가 될 것을 알 수 있다. 16번과 17번 줄 사이에

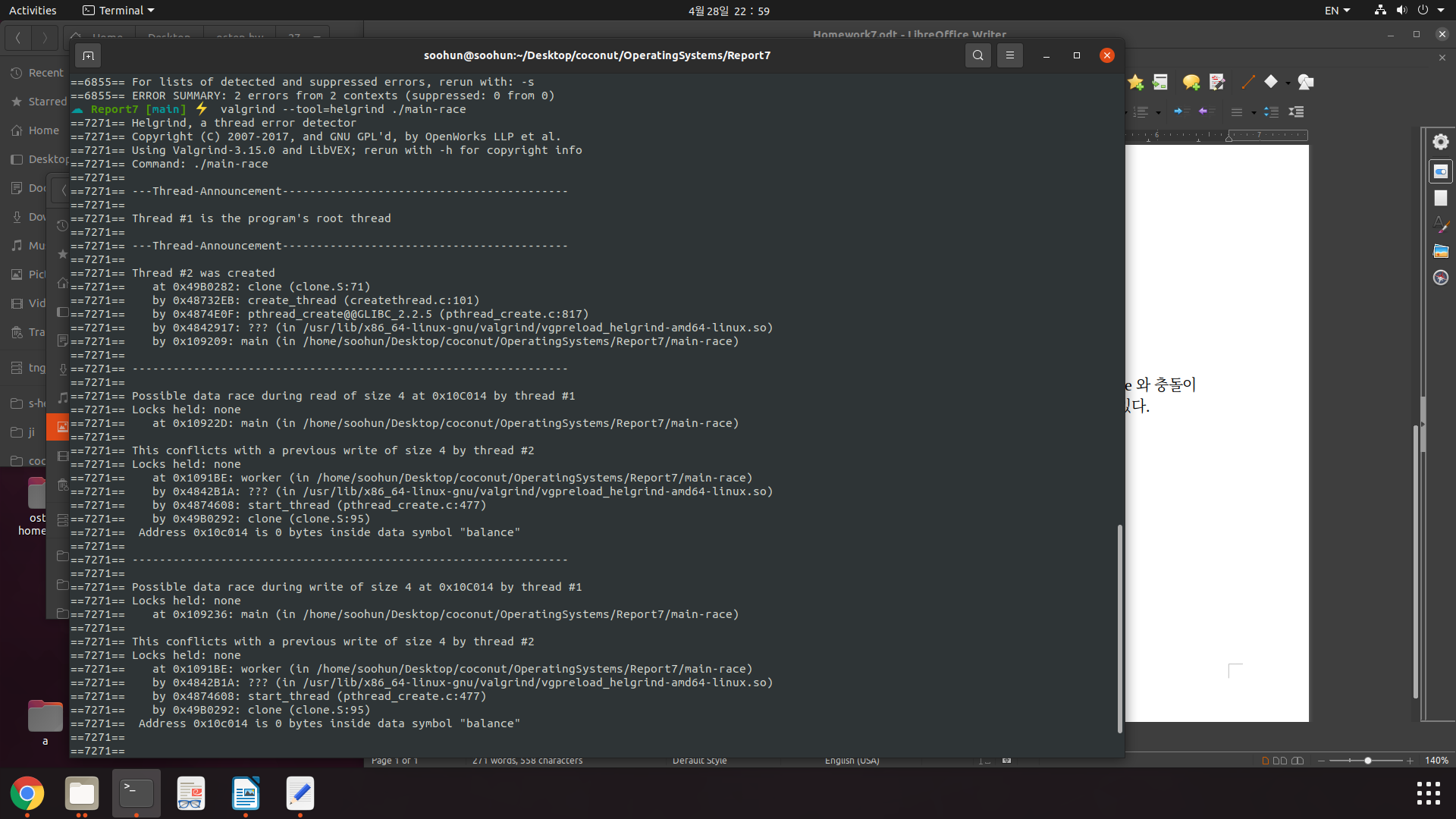
printf(“balance : %d\n”, balance);

를 추가하고 컴파일해 실행하자 역시 balance : 2 라는 결과가 나왔다.



즉, thread 안에 있는 변수가 다른 곳에서도 접근되었다는 뜻이다.

valgrind –tool=helgrind ./main-race 를 통해 확인해보았을 때 결과는 아래와 같았다.



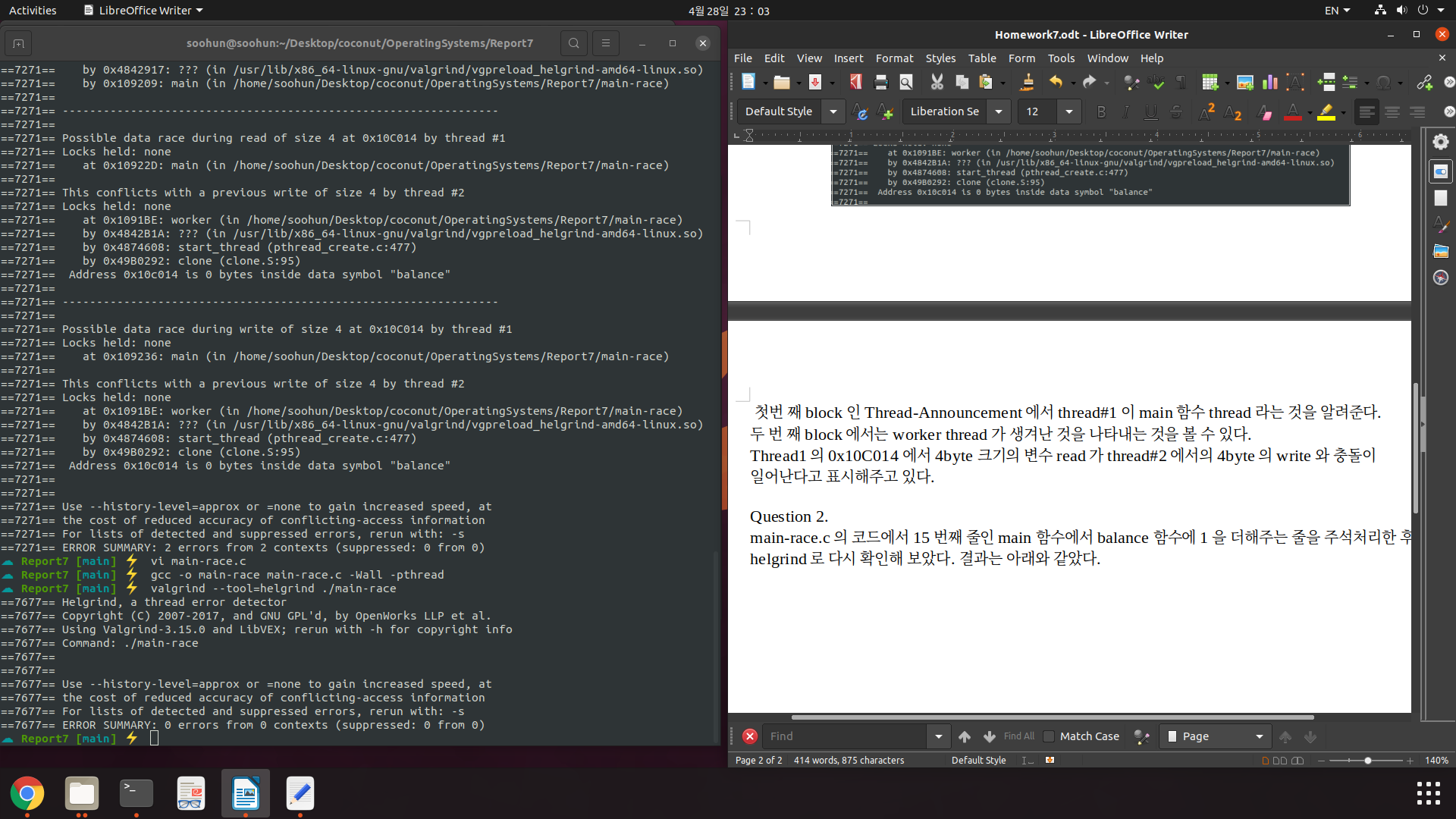
첫번 째 block인 Thread-Announcement에서 thread#1이 main함수 thread라는 것을 알려준다.

두 번 째 block에서는 worker thread가 생겨난 것을 나타내는 것을 볼 수 있다.

Thread1의 0x10C014에서 4byte 크기의 변수 read가 thread#2에서의 4byte의 write와 충돌이 일어난다고 표시해주고 있다.

Question 2.

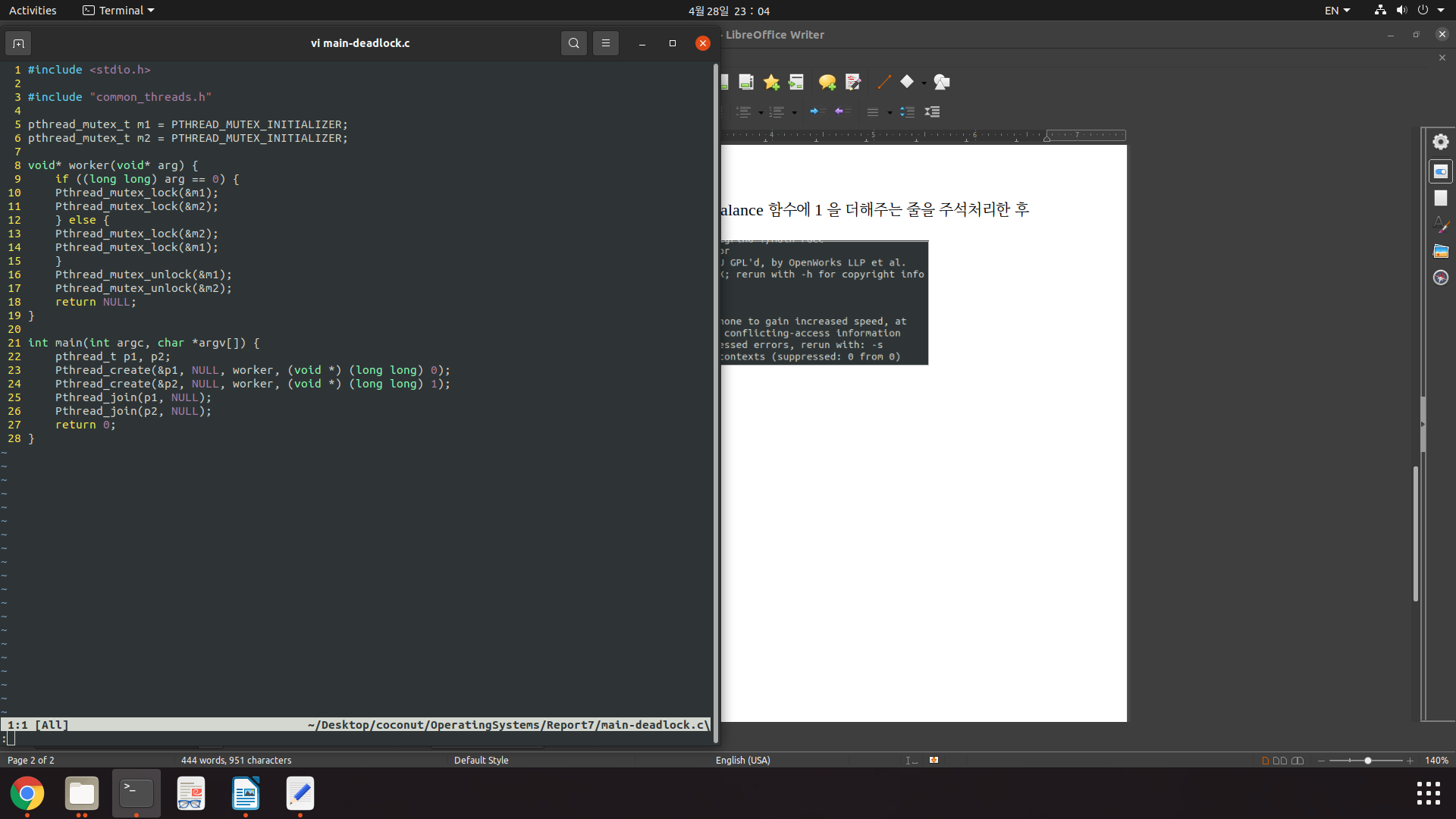
main-race.c의 코드에서 15번째 줄인 main함수에서 balance 함수에 1을 더해주는 줄을 주석처리한 후 helgrind로 다시 확인해 보았다. 결과는 아래와 같았다.



Error summary에 아무런 error도 나타나지 않았다.

Question 3.

main-deadlock.c의 코드는 아래와 같았다.



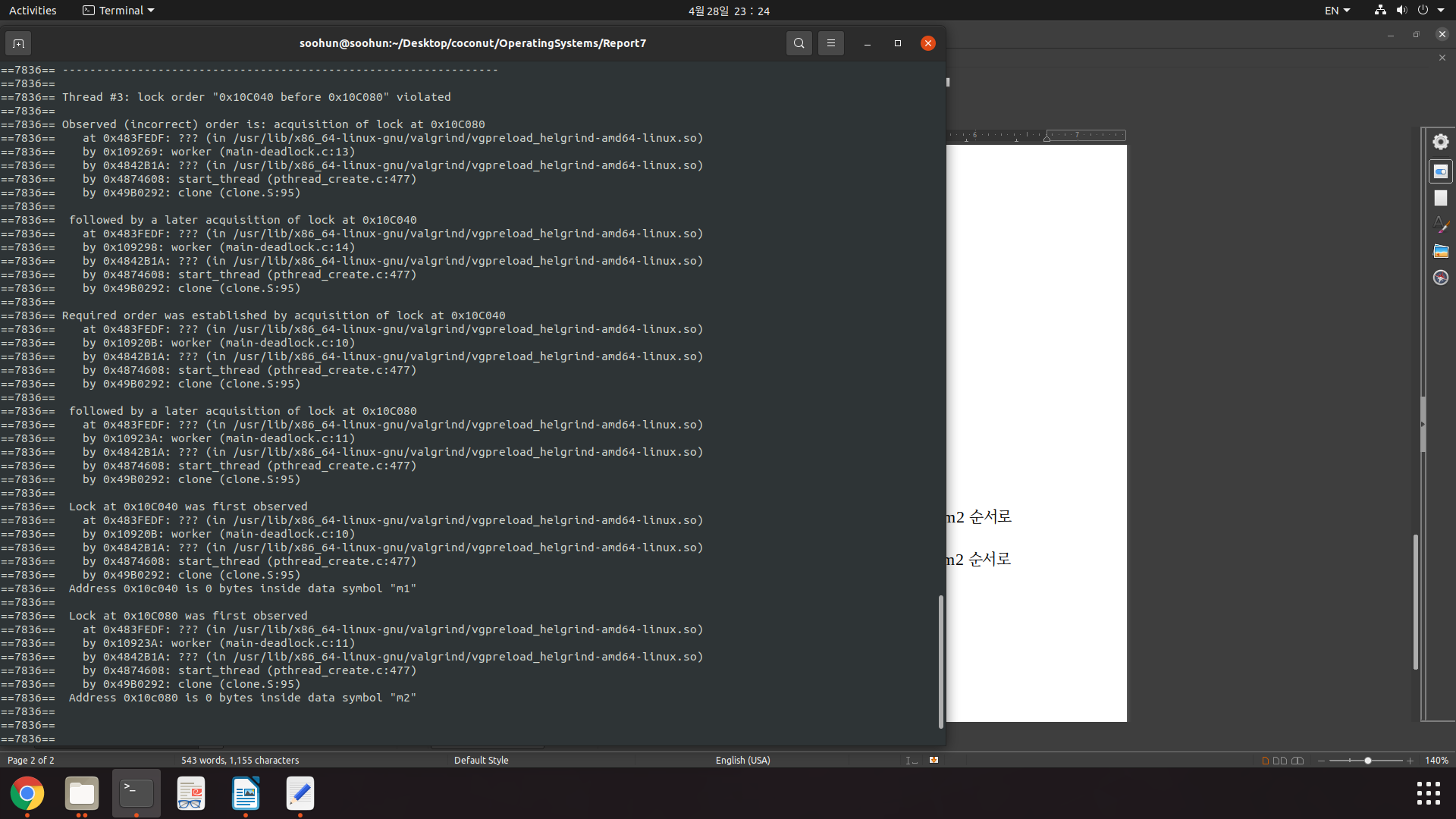
23번째 줄에서 생성한 p1 thread의 arg는 0이다. 따라서 m1, m2 순서로 lock한 후, m1, m2 순서로 unlock한다.

그 후 24번째 줄에서 생성한 p2 thread의 arg는 1이므로 m2, m1 순서로 lock한 후, m1, m2 순서로 unlock한다.

즉, 두 개의 thread가 서로를 lock하는 것을 볼 수 있다.

Question 4.

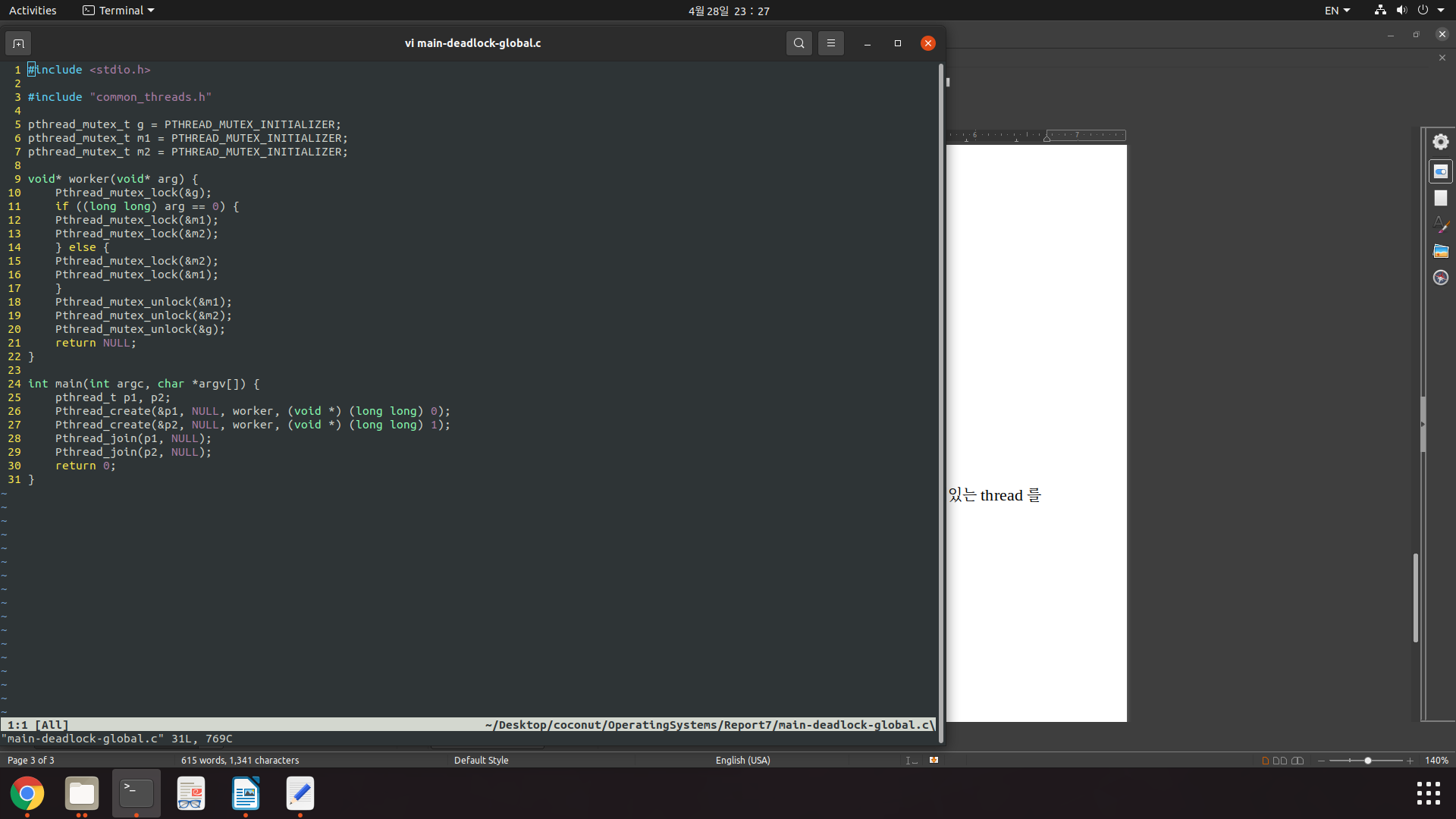
valgrind –tool=helgrind ./main-deadlock의 결과는 아래와 같았다.



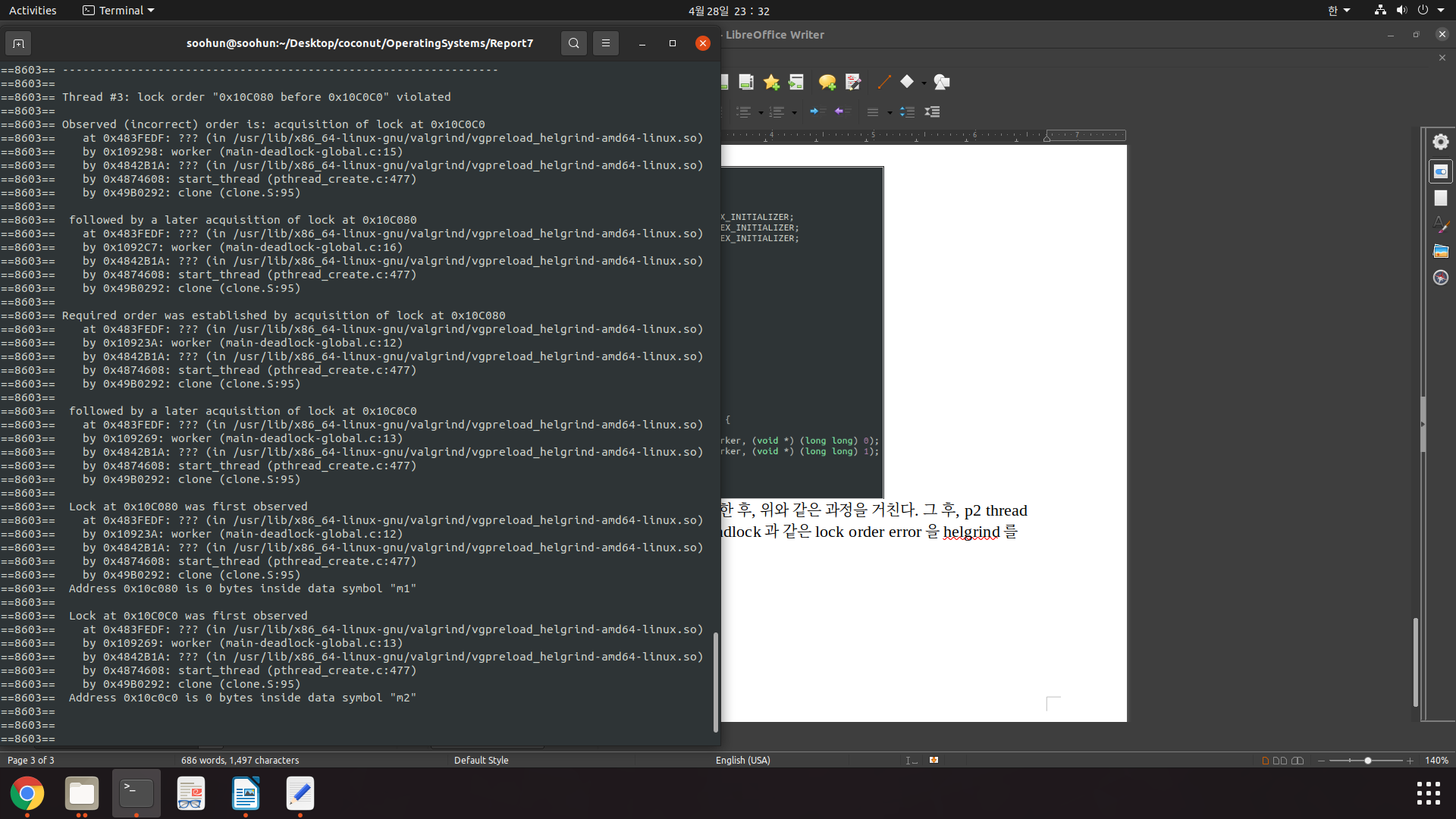
0x10C040과 0x10C080에서 lock이 실행되었지만, 밑의 두 블럭에서는 이미 lock이 되어있는 thread를 lock 했다는 order에 관한 에러를 표시했다.

Question 5.

main-deadlock-global.c의 코드는 아래와 같았다.

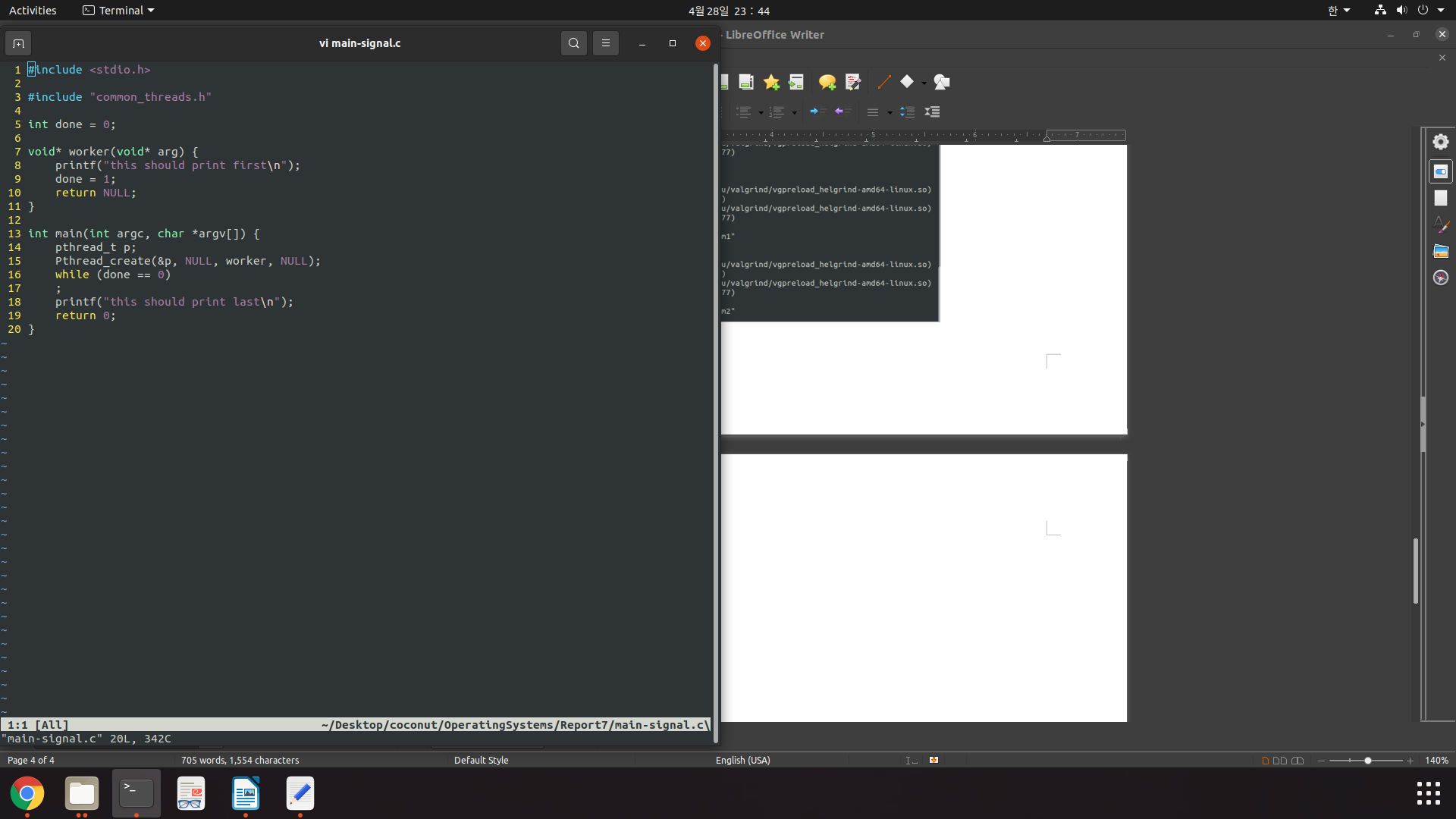


위의 코드에서는 p1 thread가 우선 g mutex의 lock을 한 후, 위와 같은 과정을 거치고 unlock한다.. 그 후, p2 thread 또한 g를 lock한 후 같은 과정을 거친 후 unlock한다. 결국 main-deadlock과 같은 lock order error을 helgrind를 통해 확인했다.



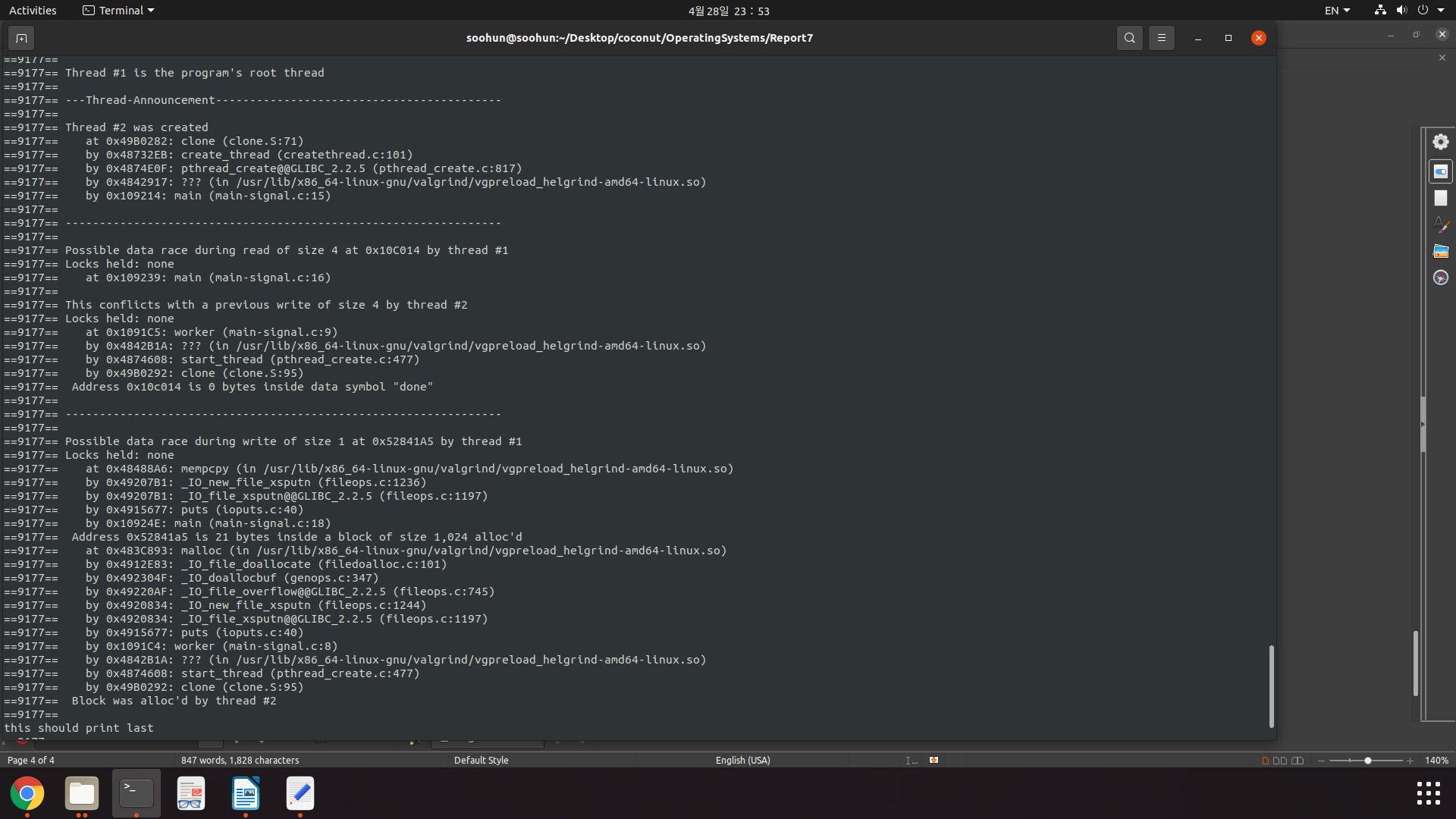
Question 6.

main-signal.c 코드는 아래와 같았다.



15번 줄에서 p thread가 만들어진다. 우선 8번 째 줄의 print문이 print되고 done 변수는 1이된다. 그 후 16번 째 줄에 while문에서 done이 1일 때, 18번째 줄의 print문을 print한 후 종료된다. 이 구조에서는 p thread가 실행되는 동안 main thread는 그저 무한 while loop를 돌게 되므로 비효율적이다. 만약 p thread가 매우 오랜 시간이 걸리는 함수라면 main thread는 기약없이 기다려야만 한다.

Question 7.



변수 done의 read와 write 의 data race와 printf 구문의 data race가 가능하다는 에러가 표시되었다. 따라서 code는 잘못되었다.

Question 8.

위의 main-signal.c 코드처럼 thread가 끝날 때까지 loop를 돌며 기다리는 것이 아니라, signal을 이용해 thread가 끝났음을 main thread가 알 수 있다. 따라서 thread가 실행되는 중에도 main thread는 자원을 낭비하지 않고 다른 일련의 코드들을 실행할 수 있으므로, 성능적인 면에서도 뛰어나고, 올바른 코드로 볼 수 있다.

Question 9.

아무런 오류도 발견되지 않았다.