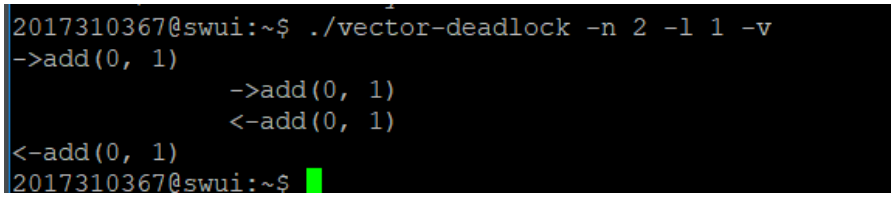
1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in vector-deadlock.c, as well as in main-common.c and related files. Now, run ./vector-deadlock -n 2 -l 1 -v, which instantiates two threads (-n 2), each of which does one vector add (-l 1), and does so in verbose mode (-v). Make sure you understand the output. How does the output change from run to run?

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "mythreads.h"

#include "main-header.h"
#include "vector-header.h"

void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}

void fini() {}

#include "main-common.c"
```

vector-deadlock 코드를 보면 for문 안에 들어가기 위해서는 dst와 src라는 두 개의 lock을 획득해야 한다. 병렬적으로 실행 중인 두 스레드가 같은 데이터를 읽고 쓸 때 어떤 명령문이 먼저 실행될 지는 결정되지 않는다. 만약 한 스레드가 데이터를 읽고 다른 스레드는 수정한다면, 스레드는 원하지 않은 값을 읽거나 수정하게 될 수 있다. 이것이 바로 스레드의 동시성의 문제이다. 이러한 동시성의 문제는 lock 기능을 통해 두 개 이상의 스레드가 동시에 접근하는 것을 막음으로써 해결할 수 있다. 그러나 이런 식으로 두 개 이상의 lock을 사용하는 것은 deadlock 문제를 일으킬 수 있다.

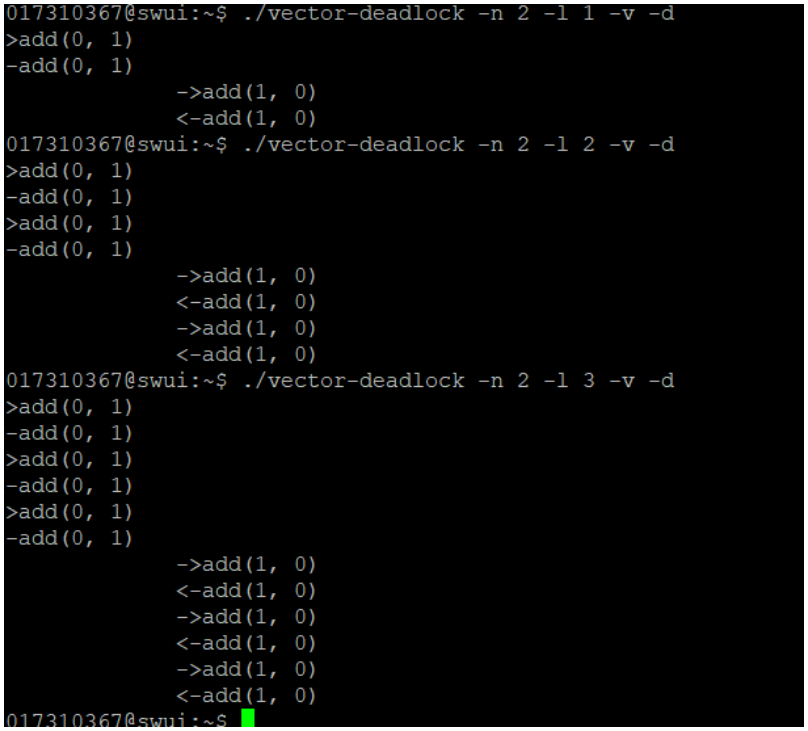
```c
} thread_arg_t;

void *worker(void *arg) {
    thread_arg_t *args = (thread_arg_t *) arg;
    int i, v0, v1;
    for (i = 0; i < loops; i++) {
        if (args->vector_add_order == 0) {
            v0 = args->vector_0;
            v1 = args->vector_1;
        } else {
            v0 = args->vector_1;
            v1 = args->vector_0;
        }

        print_info(0, args->tid, v0, v1);

        vector_add(&v[v0], &v[v1]);

        print_info(1, args->tid, v0, v1);
    }
    return NULL;
}
                                                    47,1
```

다음은 main-common의 일부분이다. main-common에서는 Main()에서 두 개의 vector를 만든 후 각각의 work()함수를 통해 스레드들이 그들에게 접근하도록 만든다. -n -l -v 같은 플래그도 이곳에서 관리한다.



vector-deadlock을 실행했을 때의 결과이다. 2개의 스레드가 2개의 vector을 lock하고 unlock하는 것을 확인할 수 있다. 첫 번째 스레드가 1번 vector를 lock하고 두 번째 스레드가 2번 vector를 lock한다. 이 후 두 번째 스레드가 2번 vector를 unlock하고 첫 번째 스레드가 1번 vector를 unlock했다. 따라서 deadlock 문제는 발생하지 않았다.

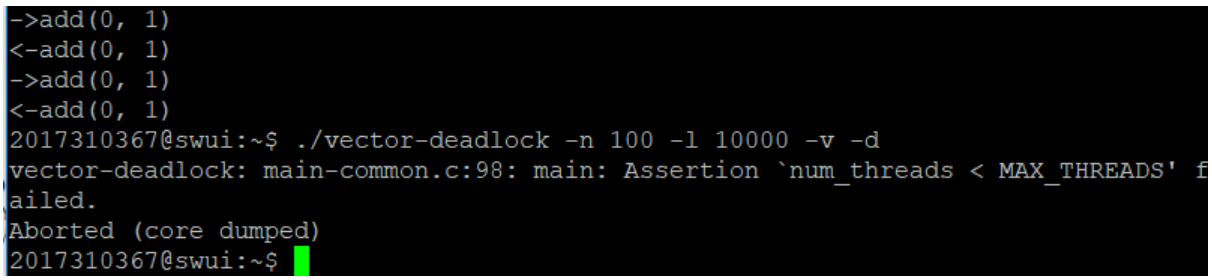
2. Now add the -d flag, and change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?



-d 플래그를 줌으로써 각각 스레드들은 다른 순서로 vector을 얻게 되었다. 스레드1이

vector_add(v1, v2)라면 스레드2는 vector_add(v2, v1)가 되는 방식이다. 즉, deadlock이 발생할 수 있는 상황을 만들어주었다. 그 이후 -l의 수를 점점 늘려보았다. 10000까지 늘렸는데도 deadlock이 발생하지 않은 것을 보면 -d를 설정했음에도 deadlock은 쉽게 발생하지 않는 것 같다.

3. How does changing the number of threads (-n) change the outcome of the program? Are there any values of -n that ensure no deadlock occurs?



-n은 스레드의 개수를 관리한다. -n을 끊임없이 늘려준 결과 드디어 deadlock 상황이 발생하였다! Deadlock이란 한 스레드가 lock을 획득하기 위해 기다리는데, 이 lock을 잡고 있는 스레드도 똑같이 다른 lock을 획득하기 위해 기다리고 있는 상태를 말한다. 스레드의 숫자가 많아지면서 각각의 스레드들이 unlock하지 않은 채로 다른 lock을 획득하려고 하자 교착상태, 즉 deadlock 상황에 들어가게 된 것이다.

lock을 얻고자 하는 스레드가 많아지면 deadlock이 발생할 확률이 높아진다. Deadlock이 절대 발생하지 않을 상황은 스레드가 한 개일 경우이다.

4. Now examine the code in vector-global-order.c. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this vector add() routine when the source and destination vectors are the same?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "mythreads.h"

#include "main-header.h"
#include "vector-header.h"

void vector_add(vector_t *v_dst, vector_t *v_src) {
    if (v_dst < v_src) {
        Pthread_mutex_lock(&v_dst->lock);
        Pthread_mutex_lock(&v_src->lock);
    } else if (v_dst > v_src) {
        Pthread_mutex_lock(&v_src->lock);
        Pthread_mutex_lock(&v_dst->lock);
    } else {
        // special case: src and dst are the same
        Pthread_mutex_lock(&v_src->lock);
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_src->lock);
    if (v_dst != v_src)
        Pthread_mutex_unlock(&v_dst->lock);
}

void fini() {}


#include "main-common.c"

~
~
~
~
~
~
```

객체 지향 프로그래밍의 핵심 개념 중 하나인 캡슐화는 외부에서 직접적으로 내부 데이터에 접근하는 것을 막는 것이다. 하지만 이러한 캡슐화에는 단점이 존재하는데 그것은 다름아닌 다른 스레드가 어떠한 명령을 수행하는지 외부에서는 전혀 알 수 없다는 것이다. 만약 두 개 이상의 스레드가 같은 메모리에 접근하려고 시도한다면 동시성의 문제가 발생할 수 있다. 이것을 막는 것이 바로 lock인데 lock을 사용할 경우 deadlock의 문제가 발생한다.

이러한 deadlock의 문제를 해결하기 위해 lock을 얻는 순서를 지정해 줄 수 있다. vector-global-order.c을 보면 dst < src일 경우 dst를 먼저 얻고 그 이후 scr를 얻도록 하고 있다. src < dst 일 경우 src를 먼저 얻고 그 이후 dst 를 얻도록 한다. 이렇게 순서를 지정해 준다면 lock을 얻으려 할 때 이미 dst를 누군가가 먼저 얻었다면 unlock이 될 때까지 기다려야 한다. Dst를 얻은 후에야 scr를 얻을 수 있기 때문에 deadlock은 발생하지 않는다. 반대의 경우도 마찬가지이다.

dst = src인 특수한 경우 어차피 두 개의 lock이 같아 deadlock 상황은 발생하지 않기 때문에 scr lcok을 얻도록 한다.

5. Now run the code with the following flags: -t -n 2 -l 100000 -d. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

```
2017310367@swui:~$ vi vector-global-order.c
2017310367@swui:~$ vi vector-global-order.c
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 100000 -d
Time: 0.03 seconds
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 1000000 -d
Time: 0.36 seconds
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 10000000 -d
Time: 3.64 seconds
```

-l로 loop를 증가시키자 더 오랜 시간이 걸리는 것을 알 수 있다. Loop가 증가하면서 lock을 얻어야 하는 상황이 더 많이 발생해 wait 시간이 길어졌기 때문으로 보인다.

```
Aborted (core dumped)
2017310367@swui:~$ ./vector-global-order -t -n 30 -l 100000 -d
Time: 0.59 seconds
2017310367@swui:~$ ./vector-global-order -t -n 40 -l 100000 -d
Time: 0.72 seconds
2017310367@swui:~$ ./vector-global-order -t -n 50 -l 100000 -d
Time: 0.86 seconds
2017310367@swui:~$
```

-n로 스레드를 증가시킨 경우에도 더 오랜 시간이 걸리는 것을 알 수 있다. 스레드의 개수가 증가하면서 lock을 얻기 위한 경쟁이 치열해져 wait 시간이 길어졌기 때문으로 보인다.

6. What happens if you turn on the parallelism flag (-p)? How much would you expect performance to change when each thread is working on adding different vectors (which is what -p enables) versus working on the same ones?

```
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 100000 -d -p
Time: 0.02 seconds
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 1000000 -d -p
Time: 0.18 seconds
2017310367@swui:~$ ./vector-global-order -t -n 2 -l 10000000 -d -p
Time: 1.74 seconds
2017310367@swui:~$ ./vector-global-order -t -n 20 -l 100000 -d -p
Time: 0.11 seconds
2017310367@swui:~$ ./vector-global-order -t -n 30 -l 100000 -d -p
Time: 0.09 seconds
2017310367@swui:~$ ./vector-global-order -t -n 40 -l 100000 -d -p
Time: 0.14 seconds
2017310367@swui:~$ ./vector-global-order -t -n 50 -l 100000 -d -p
Time: 0.11 seconds
2017310367@swui:~$
```

-p는 각각의 스레드들이 같은 vector set이 아니라 서로 다른 vector set을 갖도록 하는 플래그이다. 즉, 만약 스레드1이 vector_add(v1, v2)라면 스레드2는 vector_add(v3, v4)가 되는 방식이다. 이제 스레드들은 lock을 얻기 위해 경쟁해야 할 필요가 없어졌다. 따라서 wait 시간이 감소하였으며 더 적은 시간이 걸린 것을 확인할 수 있다.

7. Now let's study vector-try-wait.c. First make sure you understand the code. Is the first call to pthread mutex trylock() really needed? Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "mythreads.h"

#include "main-header.h"
#include "vector-header.h"

int retry = 0;

void vector_add(vector_t *v_dst, vector_t *v_src) {
  top:
    if (pthread_mutex_trylock(&v_dst->lock) != 0) {
        goto top;
    }
    if (pthread_mutex_trylock(&v_src->lock) != 0) {
        retry++;
        Pthread_mutex_unlock(&v_dst->lock);
        goto top;
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}

void fini() {
    printf("Retries: %d\n", retry);
}

#include "main-common.c"

~
~
~
~
```

vector-try-wait는 첫번째 lock을 가진 스레드가 두번째 lock을 가지려 시도했다 실패하면 첫번째 lock을 unlock하게 하는 코드이다. 즉, 하나의 lock을 붙든 채 다른 lock을 wait하지 않게 되므로 deadlock이 발생하지 않는다.

여기서 처음 불리는 pthread mutex trylock()는 반드시 필요하다. 만약 이 pthread mutex trylock()가 없다고 가정해보자. 그럼 첫번째 lock을 가지려다가 실패한 스레드가 두번째 lock을 가지는 상황이 발생할 수 있다. 첫번째 lock을 가진 스레드는 두번째 lock을 계속 얻지 못하게 된다. 그러면 deadlock 상황이 발생한다.

```
2017310367@swui:~$ ./vector-try-wait -t -n 9 -l 10000 -d
Retries: 1046960
Time: 0.85 seconds
2017310367@swui:~$ ./vector-try-wait -t -n 10 -l 10000 -d
Retries: 1197640
Time: 1.05 seconds
```

5번 문제의 vector-global-order 보다 더 적은 스레드의 개수와 loop를 가지고 있음에도 시간은 vector-global-order와 비슷하거나 더 오랜 시간이 걸린다. 즉, vector-try-wait는 vector-global-order보다 더 비효율적인 것으로 보인다. vector-try-wait의 경우 lock를 얻지 못하면 가지고 있던 lock을 버리고 처음부터 다시 lock을 얻어야 하기 때문에 더 오랜 시간이 걸리는 것으로 보인다.

```
2017310367@swui:~$ ./vector-try-wait -t -n 11 -l 10000 -d
Retries: 1765412
Time: 1.92 seconds
2017310367@swui:~$ ./vector-try-wait -t -n 12 -l 10000 -d
Retries: 2096908
Time: 2.08 seconds
2017310367@swui:~$
```

실제로 스레드의 개수를 늘리면 늘릴수록 retires의 개수가 많아진다. 스레드의 개수가 증가해 lock을 얻기 위한 경쟁이 치열해져 lock을 얻지 못하고 가지고 있던 lock을 포기하는 경우가 많아졌기 때문에 retires의 개수가 증가한 것으로 보인다.

8. Now let's look at vector-avoid-hold-and-wait.c. What is the main problem with this approach? How does its performance compare to the other versions, when running both with -p and without it?

```c
#include <stdlib.h>
#include <string.h>

#include <unistd.h>

#include "mythreads.h"

#include "main-header.h"
#include "vector-header.h"

// use this to make lock acquisition ATOMIC
pthread_mutex_t global = PTHREAD_MUTEX_INITIALIZER;

void vector_add(vector_t *v_dst, vector_t *v_src) {
    // put GLOBAL lock around all lock acquisition...
    Pthread_mutex_lock(&global);
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    Pthread_mutex_unlock(&global);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);

}

void fini() {}

#include "main-common.c"
```

vector-avoid-hold-and-wait는 lock을 하나 더 설정하여 lock을 얻기 위해서는 그 하나의 lock을 필수적으로 얻어야 한다. 그 lock을 얻어야만 두 개의 lock을 모두 얻을 수 있기 때문에 하나의 lock을 가진 채 다른 lock을 wait하는 일은 발생하지 않는다. 즉, 한 번에 모든 자원을 가지거나 아무것도 가지지 못하게 하며(hold 방지) 실행 전에 자원을 미리 할당해 버린다(wait 방지). 따라서 deadlock은 발생하지 않는다.

하지만 이러한 vector-avoid-hold-and-wait에는 치명적인 단점이 존재하는데 그것은 바로 Starvation이다. 하나의 작업이 끝날 때까지 다른 스레드들은 그 자원을 사용할 수 없으므로 자원을 효율적으로 사용할 수 없을 뿐만 아니라 Starvation 문제가 발생할 수 있다.

9. Finally, let's look at vector-nolock.c. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "mythreads.h"

#include "main-header.h"
#include "vector-header.h"

// taken from https://en.wikipedia.org/wiki/Fetch-and-add
int fetch_and_add(int * variable, int value) {
    asm volatile("lock; xaddl %%eax, %2;"
                 :"=a" (value)
                 :"a" (value), "m" (*variable)
                 :"memory");
    return value;
}

void vector_add(vector_t *v_dst, vector_t *v_src) {
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        fetch_and_add(&v_dst->values[i], v_src->values[i]);
    }
}

void fini() {}


#include "main-common.c"

~
```

vector-nolock은 lock을 사용하지 않은 채 같은 기능을 수행하는 코드이다. Fetch And Add (FAA) 명령어는 lock을 사용하지 않는 기본 원자함수이며, 추출과 덧셈 연산을 원자성을 가지고 수행한다. 즉, 원자성을 가지기 때문에 동시성의 문제가 발생하지 않게 된다. 그렇기에 다른 lock을 사용하는 코드들과 같은 기능을 수행할 수 있다.

10. Now compare its performance to the other versions, both when threads are working on the same two vectors (no -p) and when each thread is working on separate vectors (-p). How does this no-lock version perform?

```
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 100000 -d
Time: 6.06 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 1000 -d
Time: 0.06 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 10000 -d
Time: 0.65 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 100000 -d
Time: 6.32 seconds
2017310367@swui:~$ ./vector-nolock -t -n 30 -l 1000 -d
Time: 0.08 seconds
2017310367@swui:~$ ./vector-nolock -t -n 40 -l 1000 -d
Time: 0.09 seconds
2017310367@swui:~$ ./vector-nolock -t -n 50 -l 1000 -d
Time: 0.15 seconds
```

-p를 주지 않았을 때의 성능이다. 다른 프로그램들과 비교하여 특출난 점이 없다.

```
Time: 0.15 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 1000 -d -p
Time: 0.01 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 10000 -d -p
Time: 0.06 seconds
2017310367@swui:~$ ./vector-nolock -t -n 20 -l 100000 -d -p
Time: 0.19 seconds
2017310367@swui:~$ ./vector-nolock -t -n 30 -l 1000 -d -p
Time: 0.01 seconds
2017310367@swui:~$ ./vector-nolock -t -n 40 -l 1000 -d -p
Time: 0.02 seconds
2017310367@swui:~$ ./vector-nolock -t -n 50 -l 1000 -d -p
Time: 0.02 seconds
2017310367@swui:~$
```

하지만 -p를 주었을 때 다른 프로그램들과 비교하여 엄청나게 빠른 속도를 보인다.