

# SP - 5.1 Thread Programming Problem

hyeok's Log · 2022년 4월 28일

팔로우

sp



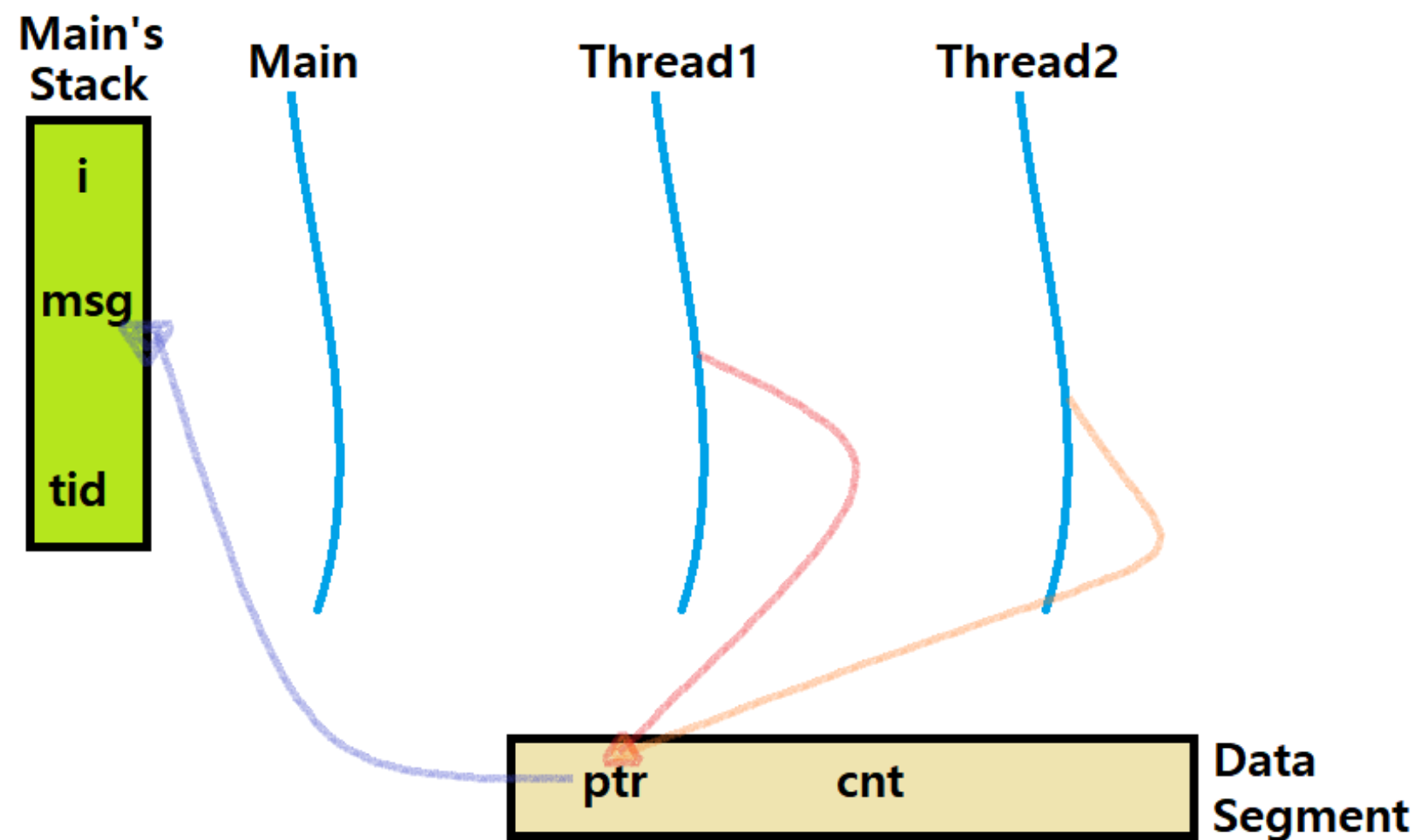
SystemProgramming



▼ 목록 보기

15/29





Chapter4까지 **Concurrent Server**에 대해 다루었다. 간단히 요약하면 다음과 같다.

- **Process-based** : Client로부터 Server에게 Connection Request가 전송될 때마다, Server는 이를 받고, fork를 띄워 Connected File Descriptor를 Child Server Process에게 넘긴다. 해당 Child Process는 Client와 Connection을 형성한다.
  - 이때, 각 프로세스는 자신만의 Address Space를 가지기 때문에, 각 Connection끼리 소통하기 위해선 **IPC(Inter Process Communication)**가 필요했다. IPC를 통해 프로세스끼리 공유 메모리를 접근하는 것이다.
- **Event-based** : 단일 프로세스 안에서 이벤트(Pending Input 여부)를 File Descriptor를 이용해 체크 후, 각 이벤트를 핸들링하는 방식이다. 하나의 프로세스가 빠르게 작업을 반복적으로 수행한다. Execution Flow가 하나이기 때문에 **Concurrent Issue**가 특별히 없다.
- **Thread-based** : 하나의 프로세스가 여러 개의 **Thread**를 띄워서 서버를 운영한다. 이들이 Process 안의 여러 Context를 Share할 수 있었다.

우리는, 이번 Chapter에서 Thread-based Server, 나아가 **Thread Programming**을 함에 있어서 발생할 수 있는 여러 **Concurrent Issue**와, 그에 대한 **Handling** 방법에 대해 다룰 것이다.

## Shared Variables

Thread는 별도의, Private한 Stack을 가진다고 했다. 하지만, 이것이 '다른 Thread로부터의 해당 Thread Stack Sharing을 막는다는 의미는 아니다. ★★'

Private란, 각 Thread가 자신의 Stack을 가졌음을 의미할 뿐, 다른 Thread로부터의 침범이 방지된다는 의미는 아니다.

따라서, 통념상론 Global Variable만 Share되고 Stack Variable은 Share되지 않을 것이라 생각하지만, 아니라는 것이다.

Thread Programming 시, 각 Thread의 Stack Variable도 Sharing의 대상이 될 수 있다.

- 즉, "Threaded C Program에서 어떤 변수들이 Share되는가?"라는 질문의 대답은 "지역변수는 Private하므로 전역변수만 공유된다."가 아니라는 것이다.

변수 A가 Shared라는 것은, 복수의 Thread가 해당 A의 인스턴스를 참조한다는 것을 말한다.

## Thread Memory Model

- 여러 Thread는 결국 하나의 단일 프로세스 위에서 돌아간다.
  - 이 '프로세스'의 **Context**는 여러 개의 Thread가 **Share**한다.
- 한편, Thread는 자신만의 **Context**도 가진다. (**Thread Context**)

## Conceptual Model

- 이론적인 시선에서 Thread와 Process의 관계는 다음과 같다.
  - 복수의 Thread가 하나의 **Process Context**를 토대로 돌아간다.
  - 각 Thread는 별도의 **Separate**된 **Thread Context**를 가진다.
    - **TID, Stack, SP(Stack Pointer), PC(Program Counter), 조건 코드, GP(Global Pointer) Register** 등을 별도로 가진다.
  - 모든 Thread는 '유지되는 **Process Context**'를 **Share**한다.
    - **Code, Data, Heap** 영역과, 프로세스의 **Virtual Memory Space** 상의 다양한 라이브러리 조각들을 공유한다.
    - 또한, **File Descriptor Table**과, 설치된 **Signal Handler**들도 공유한다. ★

## Reality

하지만, 실제로 Thread Programming을 수행, 구현하면, 다음과 같은 상황을 어렵지 않게 확인할 수 있다.

어떤 Thread가 다른 Thread를 **Read & Write** 할 수 있다.

즉, 실제로는, 이론과 다르게, 한 Thread가 다른 Thread의 **Stack Variable**도 접근할 수 있다.

- 물론, 실제로, Register 값들은 Separate하고 Protected하다. ★★
  - 그러나, Stack Variable에 대한 Access가 막히지가 않는 것이다. ★★

~> 즉, 이론과 실재가 미스매치하기 때문에, 여기서 혼란을 겪는 상황이 많다.

## Data Sharing via Indirection

아래의 예시 프로그램은 대표적인 Thread의 데이터 공유 문제를 보여준다. 주석과 함께 간단히 분석해보자.  
main Process(Thread)에 두 개의 문자열을 지역변수로 두고, 별개로 두 개의 Thread를 띄운 다음, 각 Thread가 자기 Stack에 있지 않은 main의 문자열을 띄우는 프로그램이다.

```
void *thread(void *vargp);          // Thread Routine
char **ptr;                        // 모든 Thread가 다 같이 접근 가능할 수 있는 전역변수

int main(void) {
    pthread_t tid;
    char *msgs[2] = {              // 이들은 모두 main Thread의 Local Variable들!
        "안녕, 난 foo라고 해.\n",   // 즉, main의 Stack에 존재한다.
        "안녕, 난 bar라고 해.\n",   // 그말은 즉, Peer Thread들이 접근하지 못해야한다.
    };                             // (Conceptual 관점에서 말이다)

    ptr = msgs;                    // 전역변수 ptr이 msgs라는 Local Variable의 주소를 담는다. ★
    for (long i = 0; i < 2; i++)    // i는 0 아니면 1이고,
        Pthread_create(&tid, NULL, thread, (void *)i); // i가 Argument로 넘어간다.
    Pthread_exit(NULL);             // 여긴 Thread Reaping하는 상황
}

void *thread(void *vargp) {         // Thread 0번과 1번이 이를 같이 수행한다.
    long myid = (long)vargp;
    static int cnt = 0;             // Local Variable이지만, static이므로 Data부에 들어간다.
                                    // (cnt의 Lifetime은 thread함수의 시작과 끝이다.)
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt); // myid는 0 또는 1일 것
                                    // ptr은 전역변수이고!

    return NULL;
}
// ~> 따라서, i가 0인 Thread는 foo를, 1인 Thread는 bar를 출력할 것이다.
```

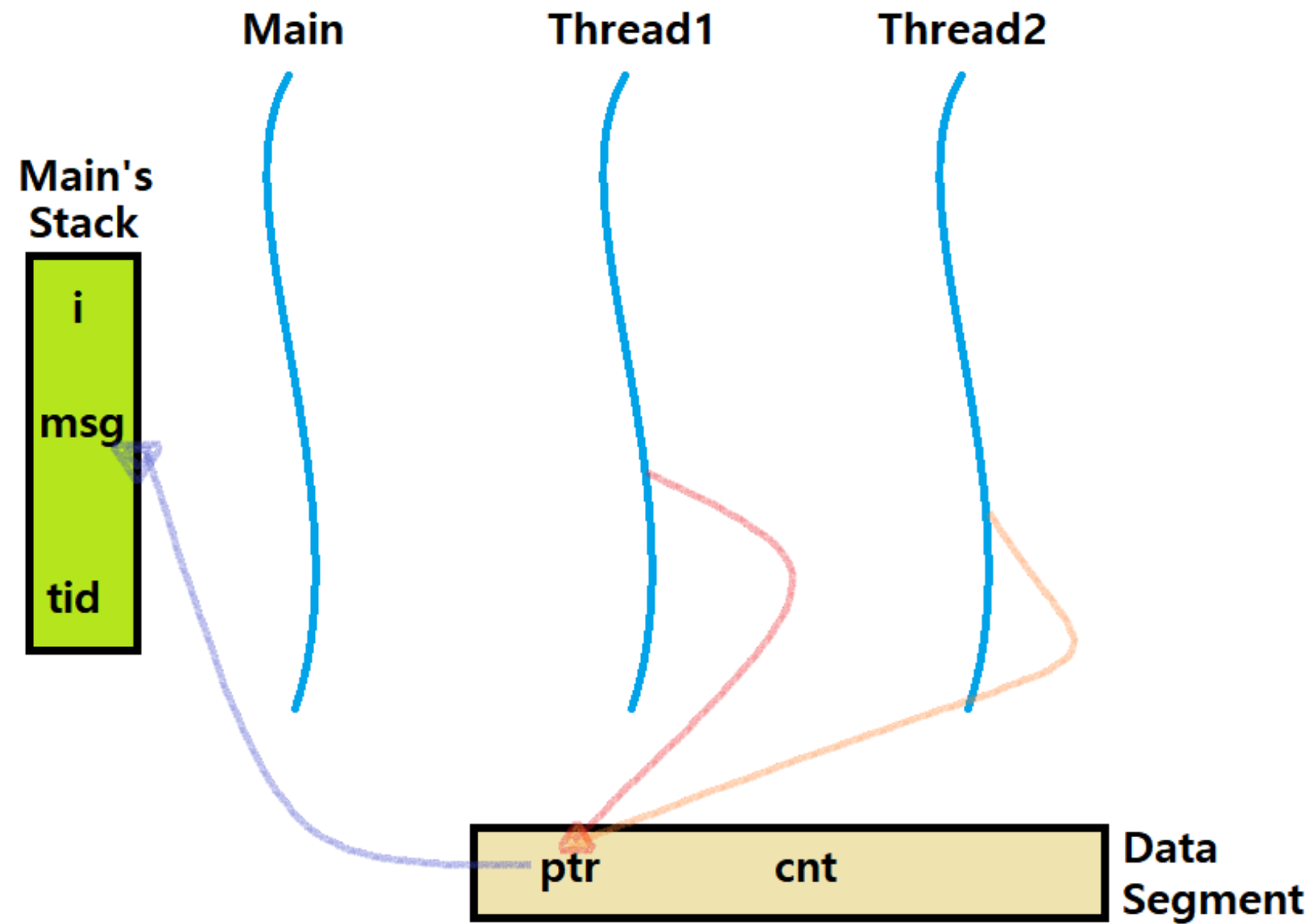
~> 각 Thread가 Data 영역에 있는 ptr을 통해서 main Thread의 Stack 영역에 있는 Local Variable인 msgs 배열에 접근할 수 있는 상황이다.

=> 단순히 접근뿐만 아니라, 충분히 수정 및 변경도 할 수 있다.

이것이 바로 **Stack Variable Sharing** 상황이다.

*Peer threads reference main thread's stack indirectly through global ptr variable.*

이러한 상황을 '**Indirection**'이라 한다. 전역 변수를 이용해 '우회적으로 접근'하므로.



# Memory Mapping of Variable Instances

- **Global Variable**

- 정의 : 함수 밖에서 정의된 변수
- 각 전역 변수에 대해, **Virtual Memory** 상에서 오직 하나의 **Instance**만 존재한다.

- **Local Variable**

- 정의 : (*static 선언 없이*) 함수 내에서 정의된 변수
- 각 Thread의 **Stack**에는 각 **Local Variable**의 **Instance**가 하나씩 존재한다.

- **Local Variable with 'static'**

- 정의 : **static Keyword**와 함께 선언된 지역 변수
- 각 static 변수에 대해, **Virtual Memory** 상에서 오직 하나의 **Instance**만 존재한다.
  - 전역변수처럼 말이다! (전역 변수와 Local Static Variable은 **Data Segment**에 존재)

*Lifetime*은 각 Thread 내에서만 존재하지만, 그 데이터 메모리 자체는 **Data Segment**에 존재한다. **딱 하나의 Instance**로만 말이다.

즉, static 변수 선언이 포함된 Thread Routine을 두 번 수행할 시, **최초 Thread Routine** 수행에서만 **Data**부에 변수를 선언한다.

- **Lifetime**이 끝나도, 그 데이터는 **Data Segment Memory**에 그대로 남아있다. 언제까지? **Main Thread(Program)**가 종료될 때까지!

```

void *thread(void *vargp);    // Thread Routine
char **ptr; → Global (Data) // 모든 Thread가 다 같이 접근 가능할 수 있는 전역변수

int main(void) {
    pthread_t tid;
    char *msgs[2] = {        // 이들은 모두 main Thread의 Local Variable들!
        "안녕, 난 foo라고 해.\n", // 즉, main의 Stack에 존재한다.
        "안녕, 난 bar라고 해.\n"  // 그말은 즉, Peer Thread들이 접근하지 못해야한다.
    };                        // (Conceptual 관점에서 말이다)

    ptr = msgs;              // 전역변수 ptr이 msgs라는 Local Variable의 주소를 담는다. ★
    for (long i = 0; i < 2; i++) // i는 0 아니면 1이고,
        Pthread_create(&tid, NULL, thread, (void *)i); // i가 Argument로 넣어간다.
    Pthread_exit(NULL);      // 여긴 Thread Reaping하는 상황
}

void *thread(void *vargp) {    // Thread 0번과 1번이 이를 같이 수행한다.
    long myid = (long)vargp;
    static int cnt = 0;        // Local Variable이지만, static이므로 Data부에 들어간다.
                                // (cnt의 Lifetime은 thread함수의 시작과 끝이다.)
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt); // myid는 0 또는 1일 것
                                // ptr은 전역변수이고!
    return NULL;
}

```

→ Local (Stack of main)

→ Local (Stack of peer thread 0)  
→ Local (Stack of peer thread 1)

→ Local static (Data)

- ptr은 Data Segment에 있다.
- tid, msgs, i(for문)는 Main Thread의 Stack에 있다.
- myid는 각 Thread의 Stack에 있다.
- cnt는 Data Segment에 있다.
- Thread는 총 3개(Main, T0, T1)가 있는데,
  - ptr (in Data) : 세 스레드가 모두 접근 가능 (전역 변수이므로)
  - cnt (in Data) : T0과 T1은 가능하지만, Main Thread는 접근 불가 (Lifetime 때문)
    - Local static 변수이므로 Lifetime이 해당 선언 Thread에 대해서만 유효하다. 해당 Thread의 Start부터 End까지 가능하다.
  - tid, i (in Stack of Main) : 당연히 Main Thread는 접근 가능, 코드 상 나머지 T0, T1에서는



접근 불가 (매우 당연한 서술)

- Indirection 되어 있지 않으므로 불가하다. ★
- *msgs (in Stack of Main) : tid, i와 다르게 Indirection 되어 있으므로 세 스레드 모두 접근 가능하다. ★★★*
- *myid (in Stack of T0 / T1) : T0, T1에서 자기 Thread의 Instance만 접근 가능. Main Thread에서는 당연히 접근 불가 (매우 당연)*

Instance	Main	T0	T1
ptr	yes	yes	yes
cnt	no	yes	yes
i	yes	no	no
msgs	yes	yes	yes
myid	no	yes	no
myid	no	no	yes

결론적으로 보면, ptr, cnt, msgs 변수는 Shared이고, tid, i, myid는 Not Shared이다.

## Synchronization Problem

우리는 이제 Thread끼리 Indirection을 통해 Stack Variable을 공유할 수 있음을 알고 있다. 그것은 좋다. 그런데, 이렇게 Variable Sharing을 할 경우, 우리는 Synchronization Error를 맞이할 수 있다. 이에 대해 알아보자. 아래의 프로그램을 보자. 주석을 함께 읽자.

```
void *thread(void *vargp); // Thread Routine

volatile long cnt = 0; // Global shared variable : Counter (in Data)
// volatile 선언은 하단에서 설명한다.

int main(int argc, char **argv) { // 이 프로그램은 Argument를 받는다.
    pthread_t tid1, tid2; // 예를 들어, ./example 10000이란 명령으로
```

```

    long niters = atoi(argv[1]);          // 프로그램을 수행했다고 가정하자.

    Pthread_create(&tid1, NULL, thread, &niters); // Thread를 두 개 띄운다.
    Pthread_create(&tid2, NULL, thread, &niters); // (T0, T1)
    Pthread_join(tid1, NULL);               // Join한다. (Reaping)
    Pthread_join(tid2, NULL);               // Joinable Threads

    if (cnt != (2 * niters))                // 결과가 20000이 아니면 잘못된 상황이다.
        printf("Shit! cnt is %ld\n", cnt);
    else                                    // 20000이 되길 기대하는 상황
        printf("Yeah! cnt is %ld\n", cnt);  // 아래의 루틴을 보면 이해 가능
    exit(0);
}

void *thread(void *vargp) {                // Thread Routine
    long i, niters = *((long *)vargp);     // 넘어온 Iteration Number에 대해,
                                           // 가정에 따르면 10000이다.

    for (i = 0; i < niters; i++)           // 10000번을 돌면서 Increment!
        cnt++;

    return NULL;
}

```

### (출력)

```

> ./example 10000
Yeah! cnt is 20000
> ./example 10000
Shit! cnt is 13051

```

~> 어떤 때는 의도한 결과가 나오는데, 어떤 때는 잘못된 결과가 나오고 있다. ★

여기서 발생한 문제는 무엇일까?

그것은 바로, **Thread Routine**에서 "cnt++;"이란 명령이 **Atomic**하다고 생각한 것이 문제의 원인이다.

전역변수 *cnt*를 각 *Thread*가 함께 접근하고 있는데, 그 *cnt*를 접근하는 명령이 *Atomic*하다고 착각한 것이다. ★

Atomic하지 않다는 것은 무슨말이고, Atomic하지 않은게 왜 문제일까? 이를 좀 더 자세히 이해하기 위해, **Main Thread의 Loop문을 Assembly Code Level로 분석**해보자.

```
for (i = 0; i < niters; i++)
    cnt++;
```

(C Source Code -> ASM Source Code)

```
    movq (%rdi), %rcx
    testq %rcx,%rcx
    jle .L2
    movl $0, %eax
.L3:
    movq cnt(%rip),%rdx    // Load      (여기는 cnt++; 부분)
    addq $1, %rdx          // Update     (여기는 cnt++; 부분)
    movq %rdx, cnt(%rip)   // Store      (여기는 cnt++; 부분)
    addq $1, %rax
    cmpq %rcx, %rax
    jne .L3
.L2:
```

~> 어셈블리어 단위로 보니, *cnt++;*이라는 명령어가 실제로는 3개의 기계어 명령으로 구성되어 있었다!!

=> 즉, "*cnt++;*"은 *Atomic*하지 않은, 3개의 *Instruction*의 *Sequence*인 것이다. ★

Atomic하지 않은 명령이 왜 문제인가? 아래를 보자.

두 개의 Thread가 Concurrent Flow 관계로 돌아가고 있다고 하자.

i (thread)	instr <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>	-	-	0
1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
1	S <sub>1</sub>	1	-	1
2	H <sub>2</sub>	-	-	1
2	L <sub>2</sub>	-	1	1
2	U <sub>2</sub>	-	2	1
2	S <sub>2</sub>	-	2	2
2	T <sub>2</sub>	-	2	2
1	T <sub>1</sub>	1	-	2

의도에  
부합하는 상황

~> 초록색은 Thread0의 명령 시퀀스, 보라색은 Thread1의 명령 시퀀스이다. 그리고, 초록과 보라 사이의  
흰색 칸은 Context Switch를 의미한다. ★  
=> (운이 좋게도) 문제가 없는 상황이다.

'Shit !' Situation1

i (thread)	instr <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>	-	-	0
1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
2	H <sub>2</sub>	-	-	0
2	L <sub>2</sub>	-	0	0
1	S <sub>1</sub>	1	-	1
1	T <sub>1</sub>	1	-	1
2	U <sub>2</sub>	-	1	1
2	S <sub>2</sub>	-	1	1
2	T <sub>2</sub>	-	1	1

'Shit !' Situation2

i (thread)	instr <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>	-	-	-
1	L <sub>1</sub>	0	-	0
2	H <sub>2</sub>	-	-	-
2	L <sub>2</sub>	-	0	-
2	U <sub>2</sub>	-	1	-
2	S <sub>2</sub>	-	1	1
1	U <sub>1</sub>	1	-	-
1	S <sub>1</sub>	1	-	1
1	T <sub>1</sub>	-	-	1
2	T <sub>2</sub>	-	-	1

~> 이번엔 Incorrect Ordering 상황이다. 각 Thread의 "cnt++;" 수행이 다 끝나쳐지기 전에, 도중에  
Context Switch가 일어났다. ★

~~> 좌측을 보자. T0에서 Load하고, Update까지 해놓았다. 그러다 **도중에 Context Switch**가 일어났다. 이제 T1의 차례이다. T1이 cnt를 자신의 rdx Register에 Load한다. 그 다음 곧바로 다시 Context Switch가 일어났다. 이제 T0의 Store가 일어난다. 그러고 다시 곧바로 Context Switch가 일어났다. 이번엔 T1이 Update해야한다. 자신의 rdx를 Increment한다. 이때, 이 값은? 그렇다. 1이다. 따라서 **1을 다시 cnt(현재 T0에 의해 1이 된)를 덮어씌운다.** 최종적으로 **cnt가 2가 아니라 1인 상황**이다. ★★

~~> 우측도 보자(그림이 약간 잘못되었다). cnt가 0인 상황에서, T0의 L1이 수행된다. 이어서 Switch가 일어나고, T1의 L-U-S가 모두 수행된다. 당근 cnt는 1일 것이다. 이어서 Switch가 일어나고, T0의 나머지 U-S가 수행되는데, 이때, **T0의 rdx에는 0이라는 값이 있었으므로, 1을 cnt에 덮어씌운다.** 즉, 역시나 **최종적으로 cnt가 2가 아니라 1인 상황**이다. ★★

=> 즉, Sequentially Consistent **Interleaving** 상황에 따라, 의도와는 동 떨어진 결과가 나타날 수 있는 것이다.

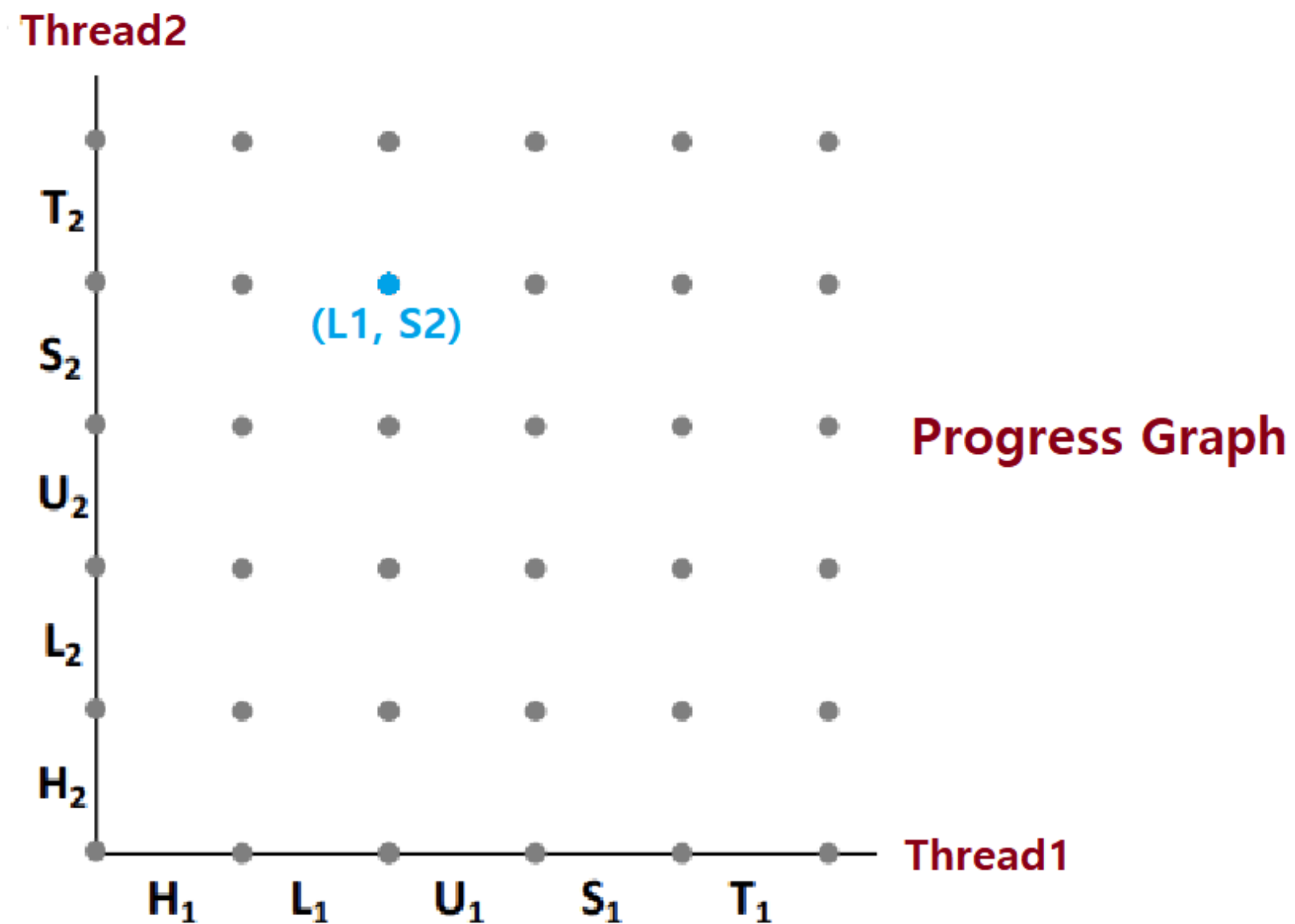
Shared Variable을 다루는 Thread의 Routine은 Atomic해야한다. 그렇지 않으면, 위와 같이 수행 도중의 Context Switch에 의해 의도가 왜곡될 수 있다.

※ cnt 전역 변수를 volatile로 선언하는 것의 효과 : Main Memory **DRAM의 Object**와 (CPU 옆의) **Cache에 있는 Object**(DRAM에서 가져온)가 서로 동기화되도록 volatile 선언을 한다.  
~> 즉, Main Memory와 Cache의 Version 차이를 방지하는 것이다.  
~> 무슨말이냐 하면, **Concurrent Programming** 시에는 어느 Thread(Process)가 먼저 해당 전역 변수에 접근할지 모르기 때문에, **Cache와 실제 DRAM 상의 Variable Object Value가 서로 다를 수 있다.** volatile 선언을 하면, 사실상 Cache가 없는 효과를 내어, **Version & Concurrency Issue**를 막는 것이다. ★

## Progress Graph

우리는 이러한 상황을, '**Progress Graph**'로 좀 더 자세히 분석할 수 있다. 연재 초반에 다룬 Process Graph와는 다른 것이다. 주의하라.

- **Progress Graph : Concurrent Flow** 관계의 두 Thread의 '자세한 명령 진행 상황(Discrete Execution State Space)'을 나타낸다.
  - 하나의 State를 나아가는데, 여러 갈래의 Path가 있다.
  - 각 **Axis**는 Thread의 순차적 명령 수행 순서를 나타낸다.
  - 각 **Point**는 **Possible Execution State**를 나타낸다.
    - **ex)** (L1, S2) : Thread1이 Load까지의 명령을, Thread2가 Store까지의 명령을 수행해놓은 State를 나타낸다.



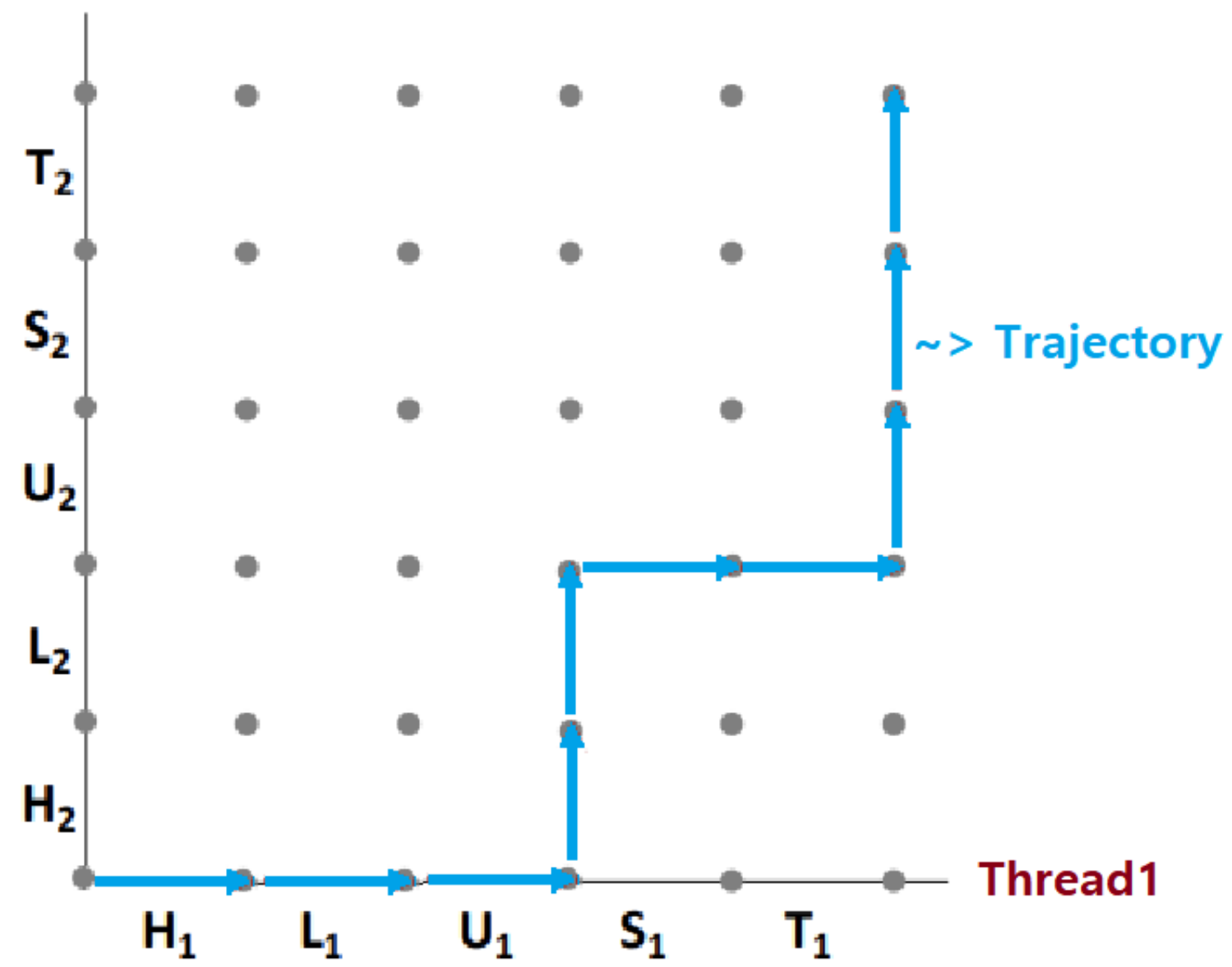
- **Trajectory : State Transition의 Sequence**를 우리는 'Trajectory'라고 부른다.
  - **Concurrent Flow**의 두 Thread의 가능한 수행 상황을 나타낸다.
- 현재, 이 예시에서, 각 Thread는 'Header->Load->Update->Store->Tail'의 과정을 거친다.

- 이 중, '**Load->Update->Store**'의 과정이 **Atomic**해야, 우리가 의도하는 프로그래밍이 되는 것이다.
  - 이러한, '프로그래머의 의도에 맞는 수행이 이뤄지기 위해 **Atomic**해야하는 명령 시퀀스'를 '**Critical Section**'이라고 부른다. ★★
  - 두 **Critical Section**이 이루는 '(교차)영역'을 우리는 '**Unsafe Region**'이라고 한다. 침범이 일어나면 안되는 지역이기 때문에 이러한 이름이 붙었다. ★★
    - *Trajectory가 Unsafe Region을 침범하면 20000이라는 결과가 나오지 않는 것이고, 구역을 침범하지 않으면 20000이 나오는 것이다. ★*

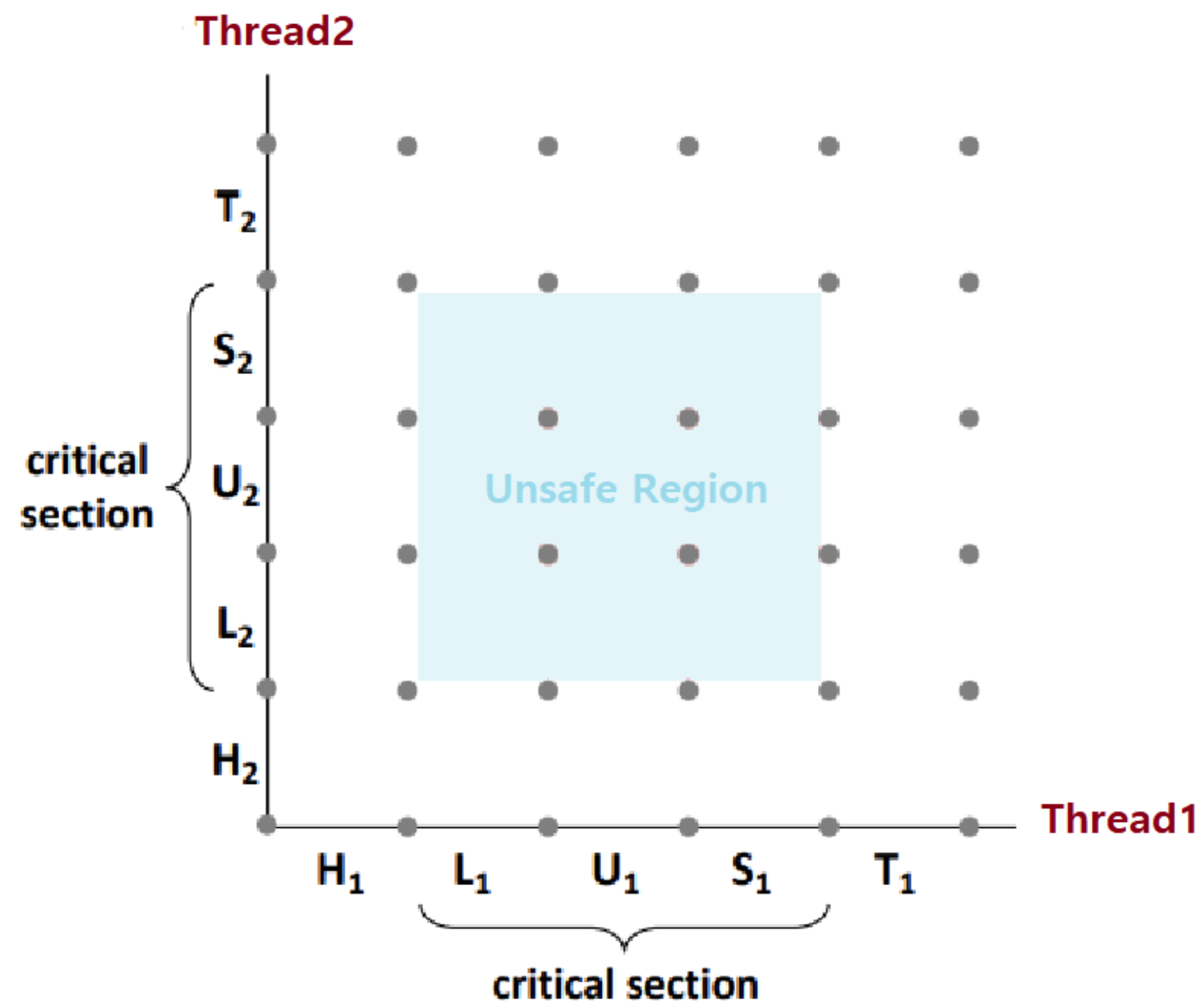
*Instructions in Critical Section should not be interleaved!!*

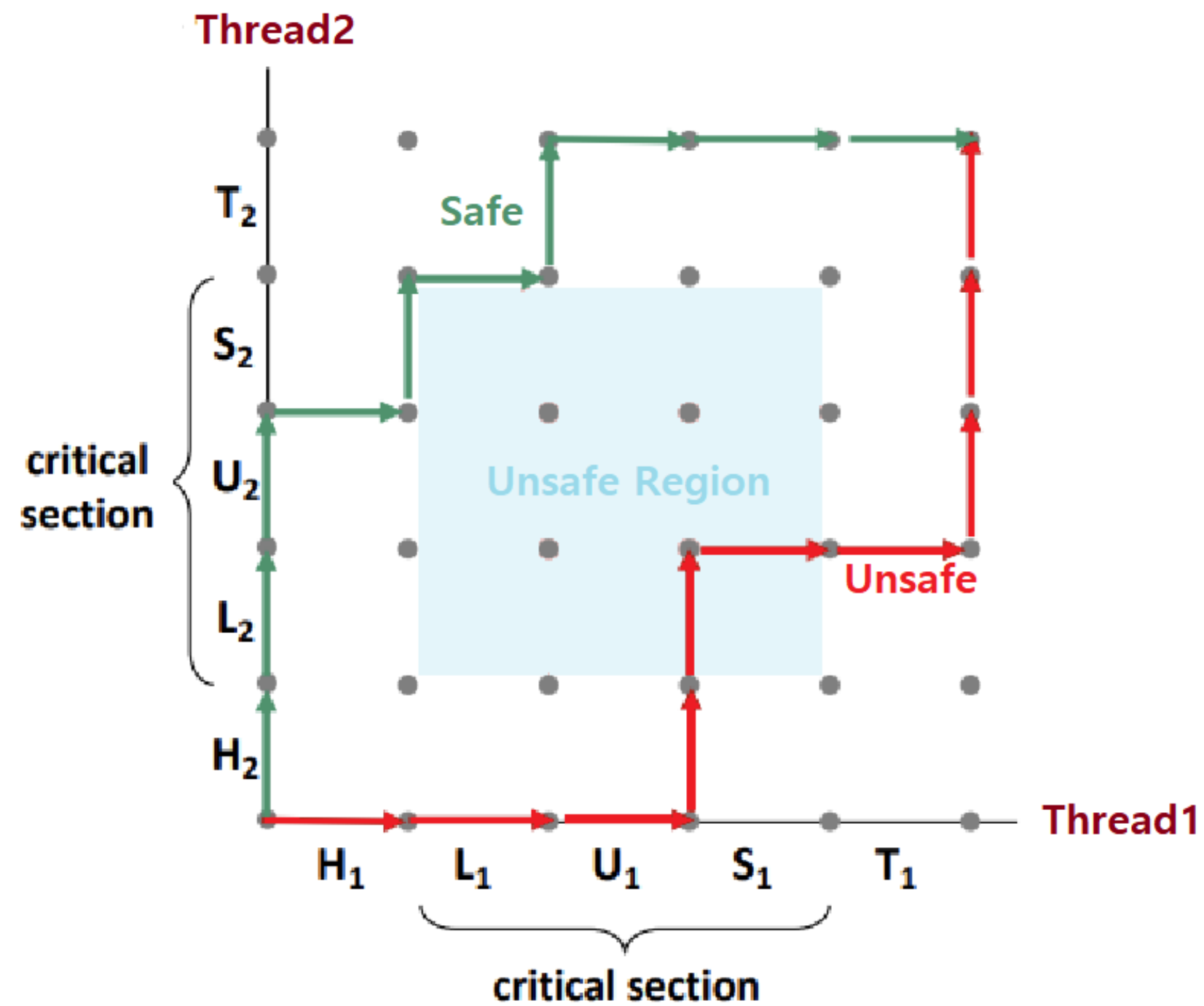
- **Unsafe Region**을 침범하지 않는 **Trajectory**를 '**Safe Trajectory**'라고 한다.
  - 'Safe Trajectory'는 곧 '**Correct Trajectory**'이다. ★

Thread2







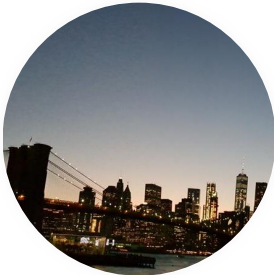


- Unsafe Region을 거치지 않는 Trajectory만이 Correct하다.
  - 이를 위해선, Load->Update->Store로 이어지는 **Critical Section**이 Atomic하게 수행되어야 한다. ★★★

즉, 이 **Critical Section**을 보호해주는 장치가 필요하다.

이를 '(Thread) Synchronization'이라 한다.


금일 포스팅은 여기까지이다. 우리는 다음 포스팅에서 설명한 동기화 문제에 대한 해결책에 대해 알아볼 것이다.



hyeok's Log

팔로우






이전 포스트

SP - 4.3 Thread-based Concur...

다음 포스트

SP - 5.2 Semaphore Synchroni...



0개의 댓글

댓글을 작성하세요

댓글 작성

