CS 개념정리: thread와 lock

thread의 개념 및 공유 변수 접근을 방지하기 위한 lock

12223682정유진

process와 thread

process, thread 개념

process

실행 중인 프로그램

```
for(int i=0; i<5; i++){
  printf("%d", i);
}
```

thread

프로그램 내 실행흐름

i=1일 때의 실행흐름

thread

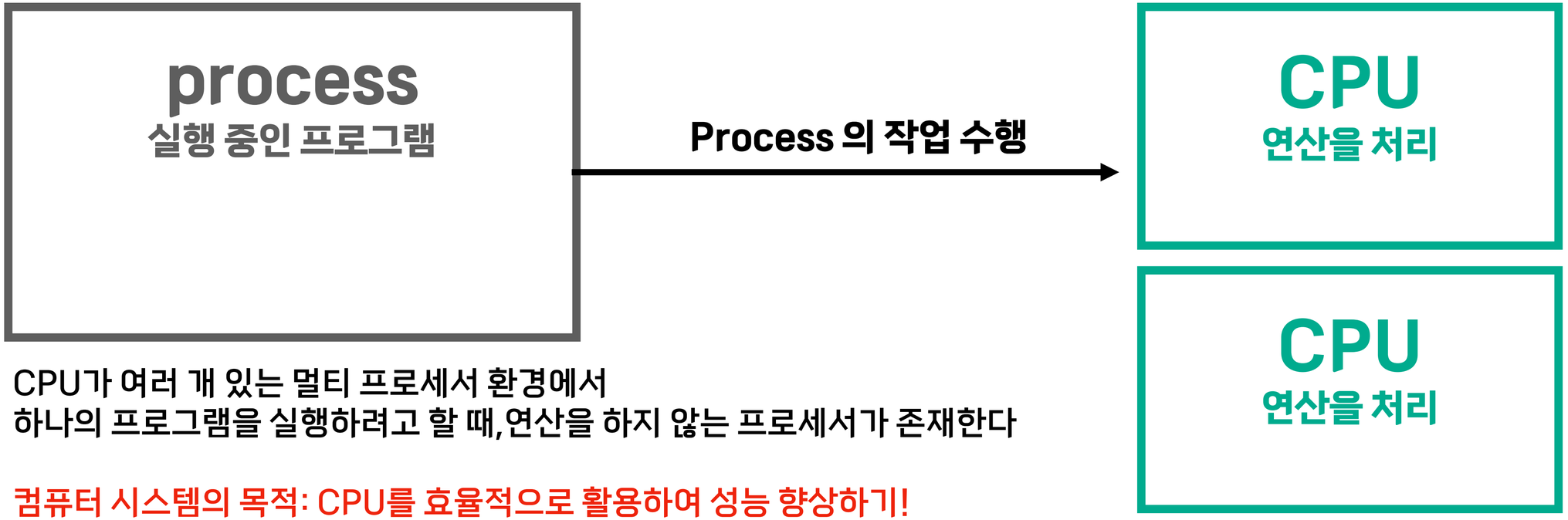
프로그램 내 실행흐름

i=2일 때의 실행흐름

process와 thread

thread를 사용했을 때의 장점

1. 병렬적으로 더 빠른 시간 내에 작업을 처리할 수 있다



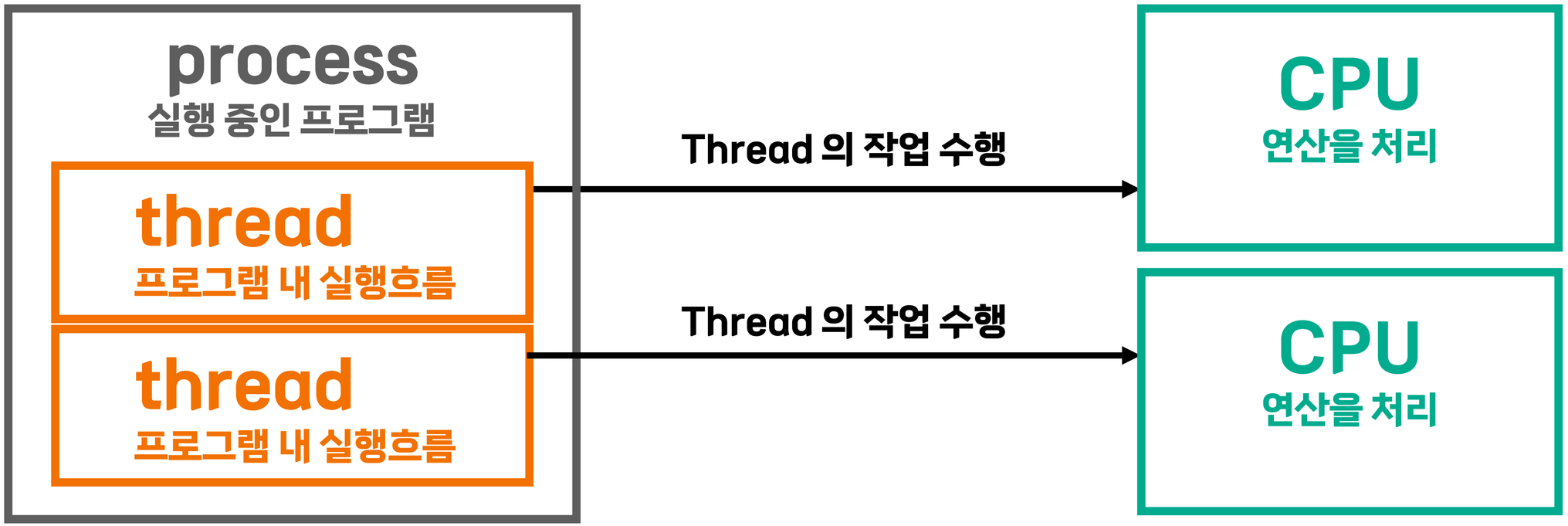
CPU가 여러 개 있는 멀티 프로세서 환경에서
하나의 프로그램을 실행하려고 할 때,연산을 하지 않는 프로세서가 존재한다

컴퓨터 시스템의 목적: CPU를 효율적으로 활용하여 성능 향상하기!

process와 thread

thread를 사용했을 때의 장점

1. 병렬적으로 더 빠른 시간 내에 작업을 처리할 수 있다

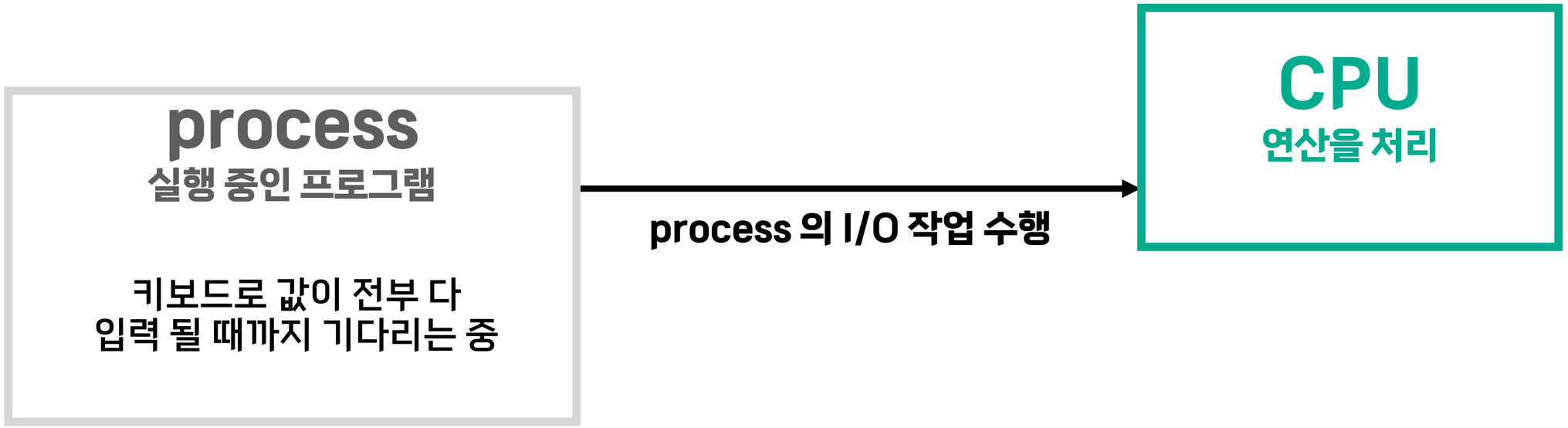


process와 thread

thread를 사용했을 때의 장점

2. I/O작업을 수행하는 경우, 멈추지 않고 다른 동작을 진행할 수 있다

외부장치(키보드, 마우스 등)와 상호작용하는 작업
CPU에 비해 훨씬 느리다. 최소 수천 배!

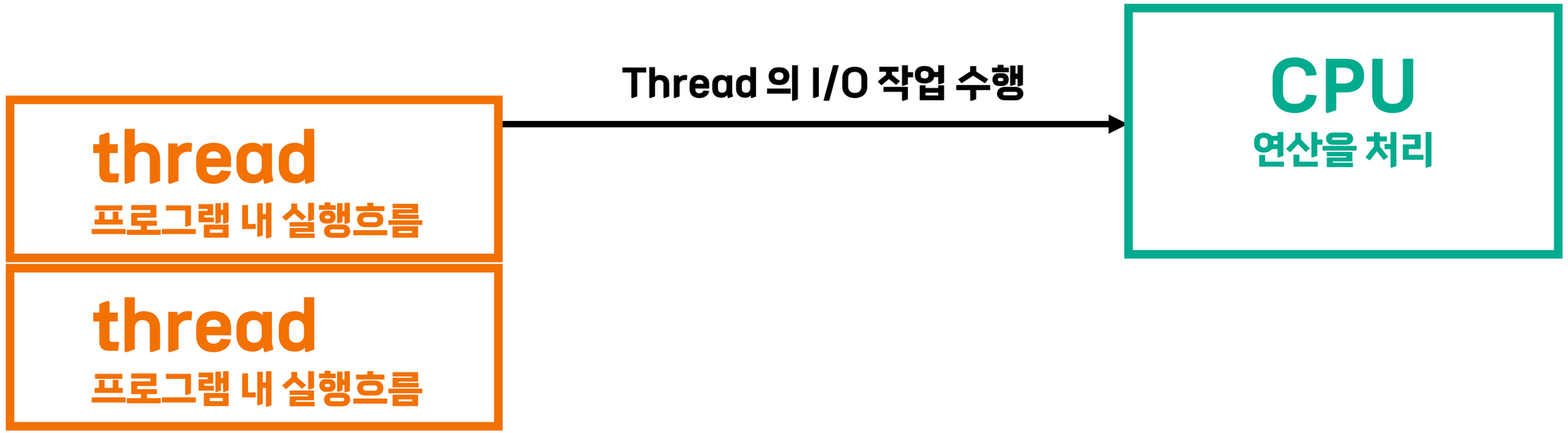


process와 thread

thread를 사용했을 때의 장점

2. I/O작업을 수행하는 경우, 멈추지 않고 다른 동작을 진행할 수 있다

외부장치(키보드, 마우스 등)와 상호작용하는 작업
CPU에 비해 훨씬 느리다. 최소 수천 배!

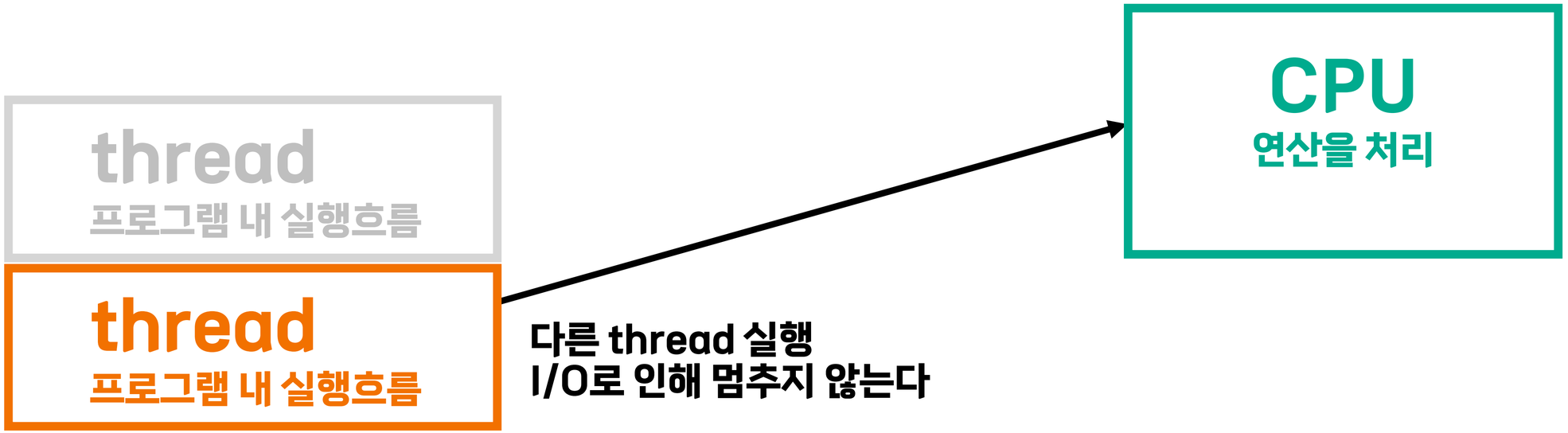


process와 thread

thread를 사용했을 때의 장점

2. I/O작업을 수행하는 경우, 멈추지 않고 다른 동작을 진행할 수 있다

외부장치(키보드, 마우스 등)와 상호작용하는 작업
CPU에 비해 훨씬 느리다. 최소 수천 배!



process와 thread

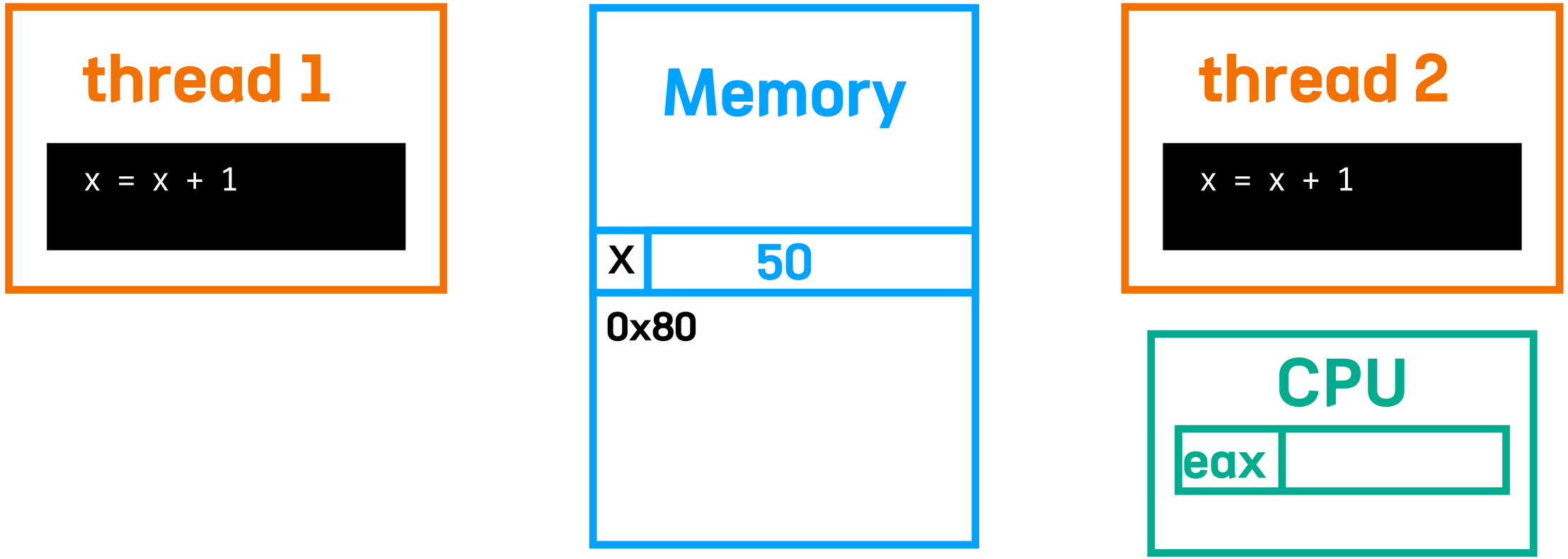
thread와 공유 변수

한 process 안의 thread들은 같은 메모리 공간을 공유한다

process와 thread

thread와 공유 변수

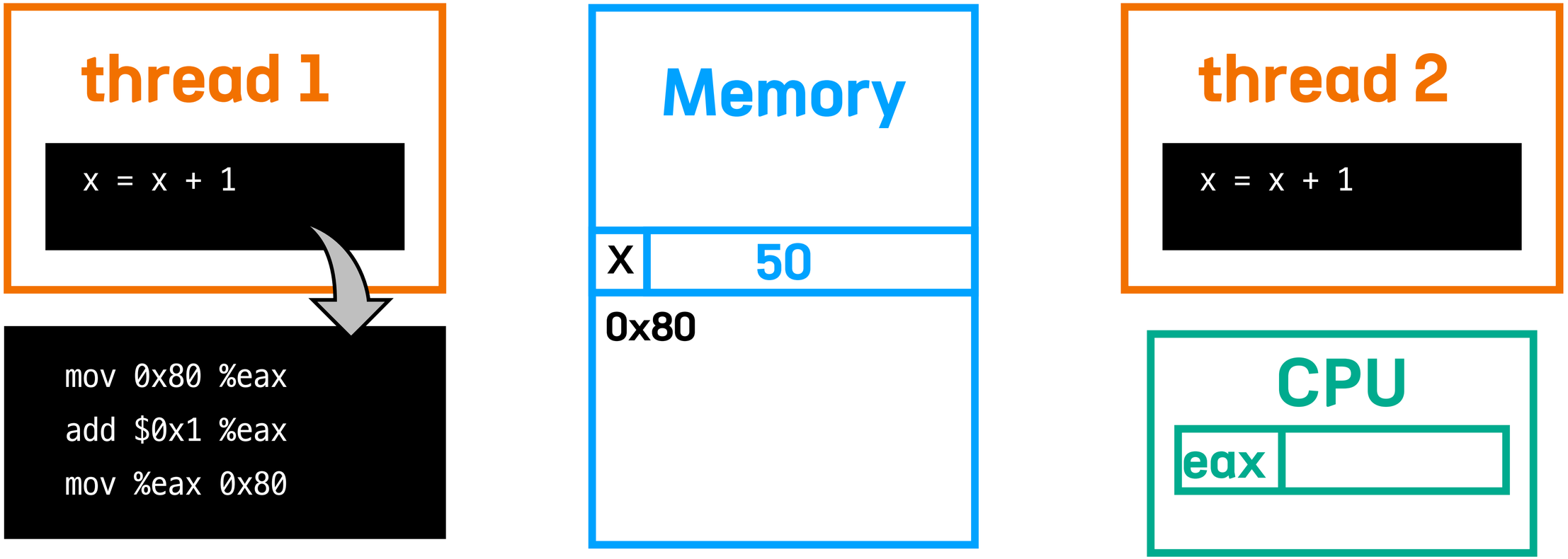
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

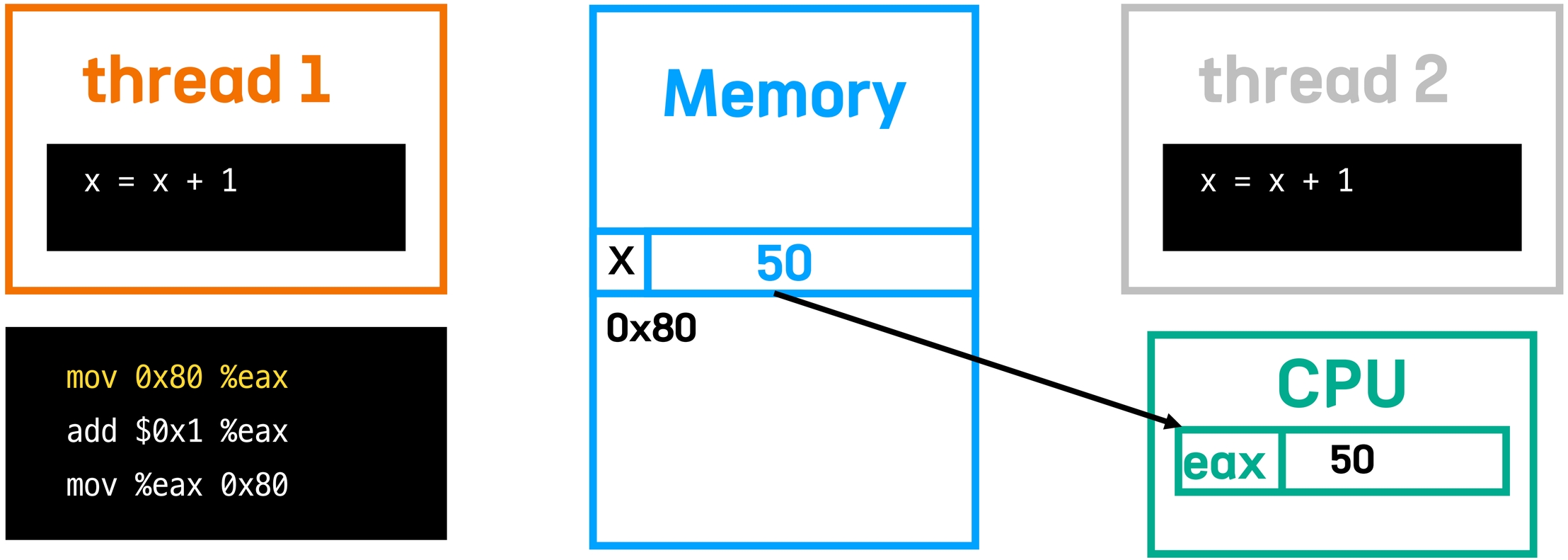
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

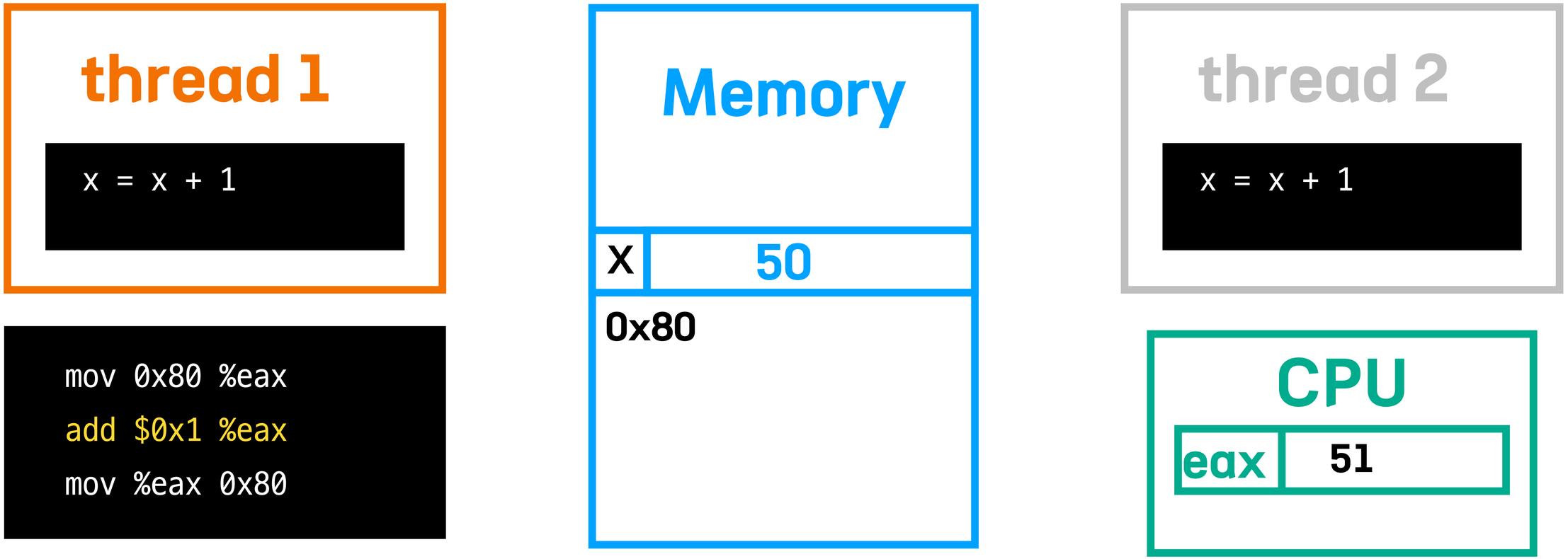
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

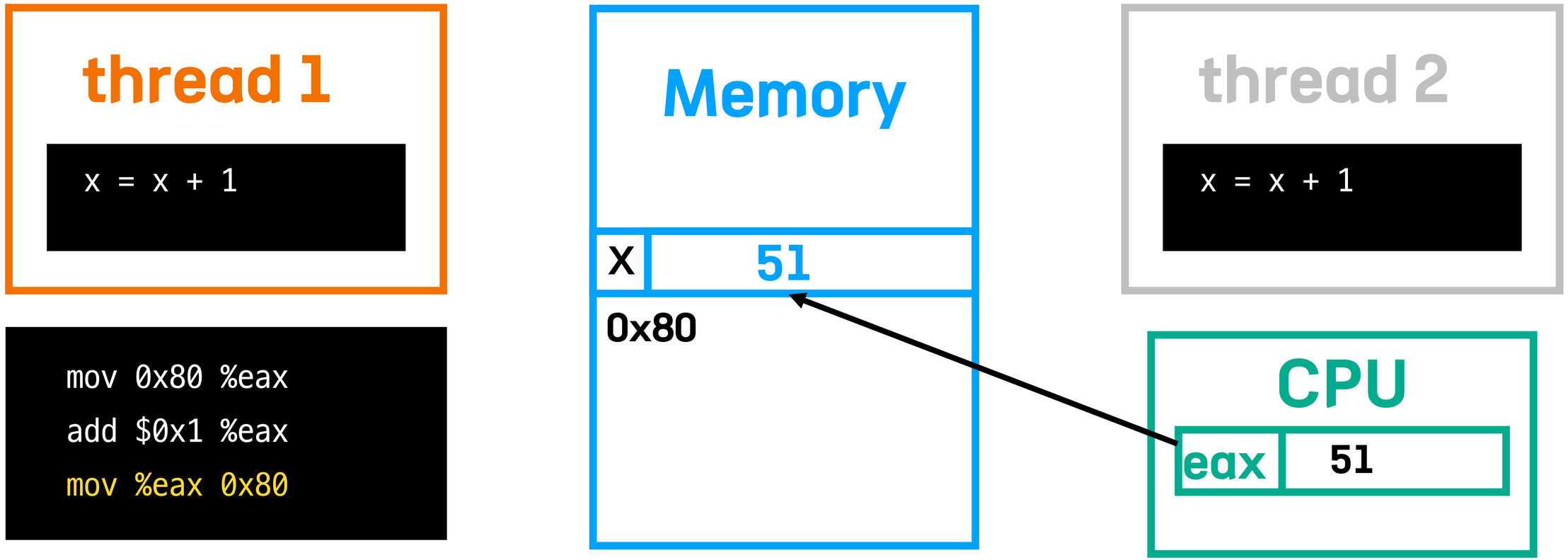
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

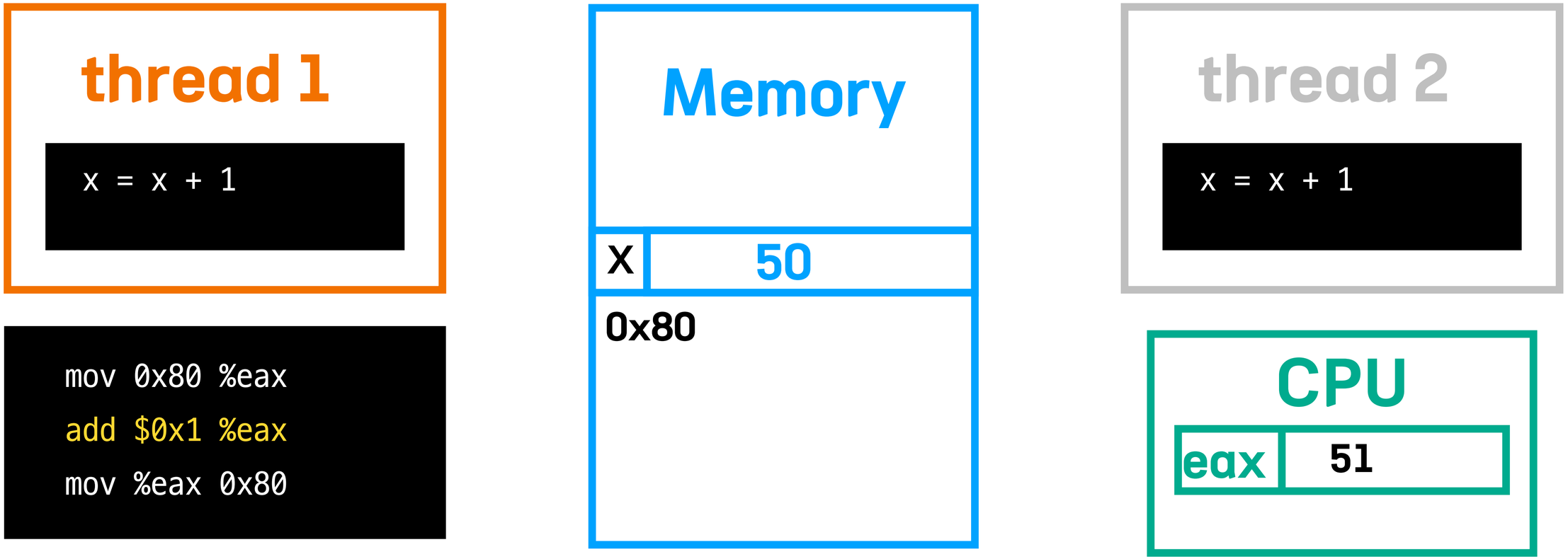
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

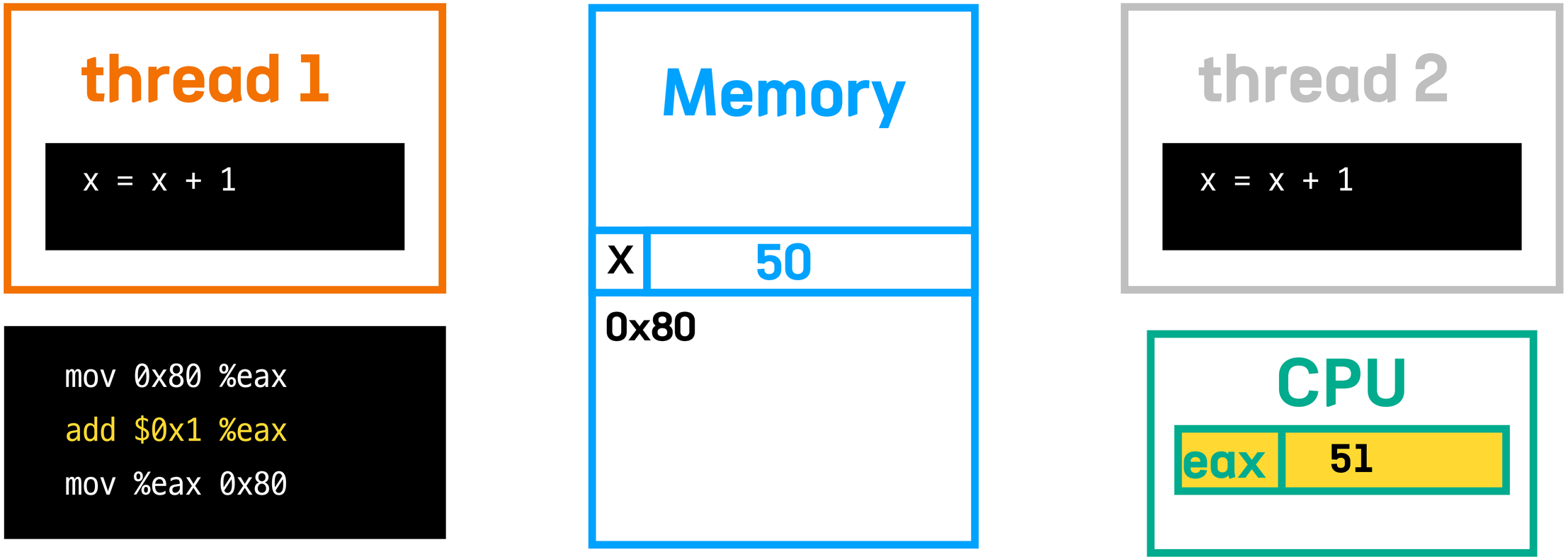
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

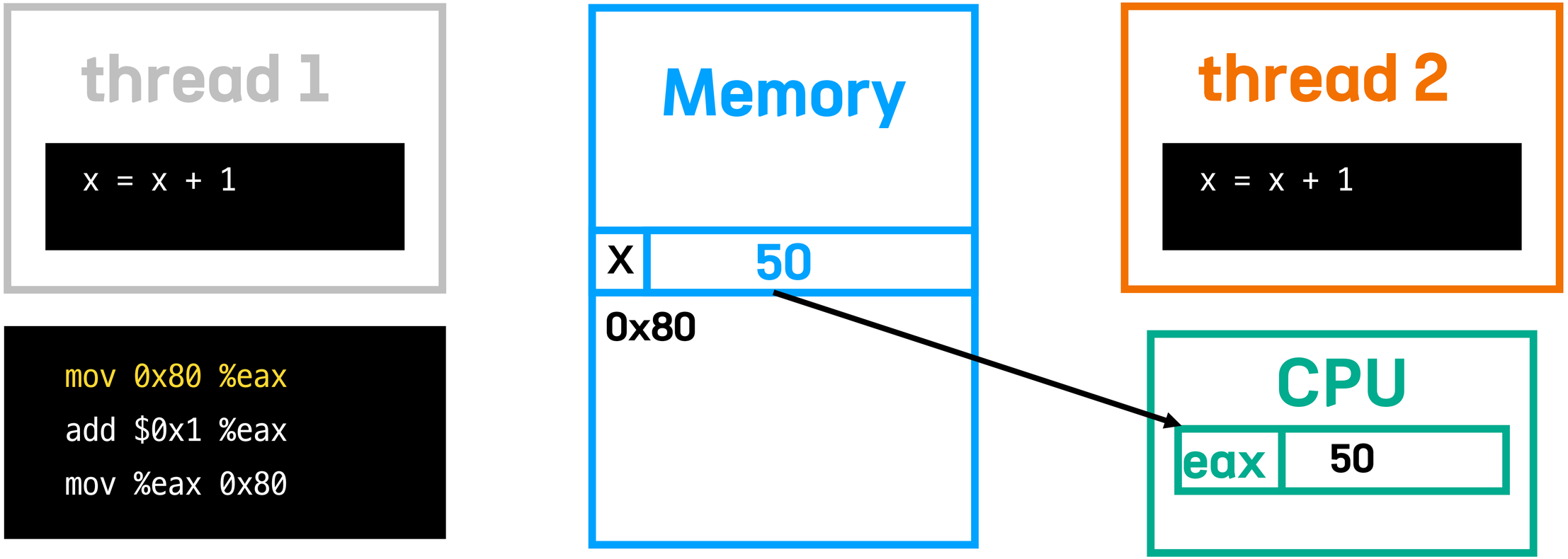
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

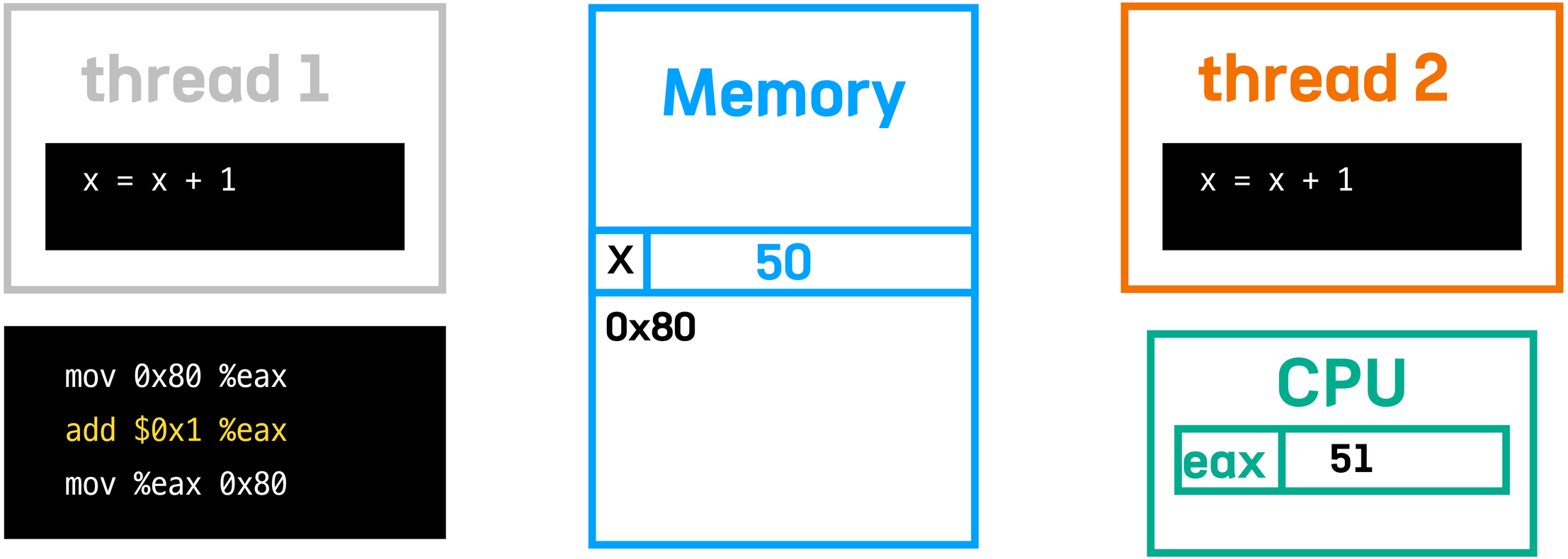
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

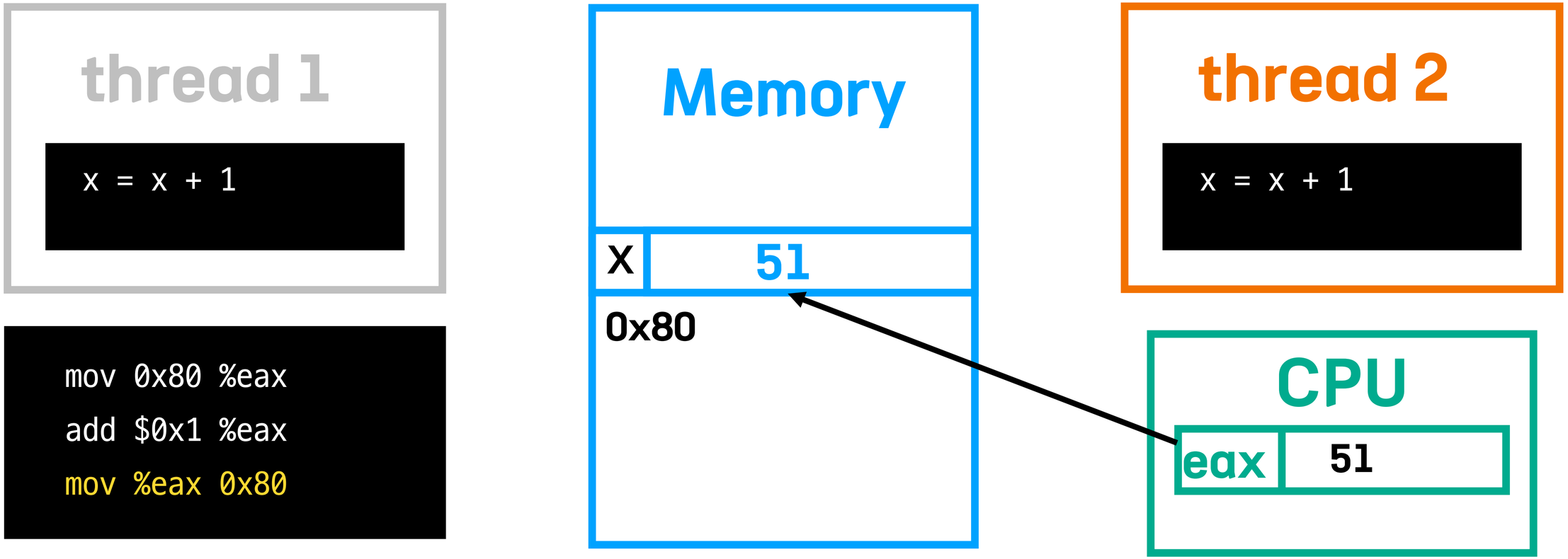
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

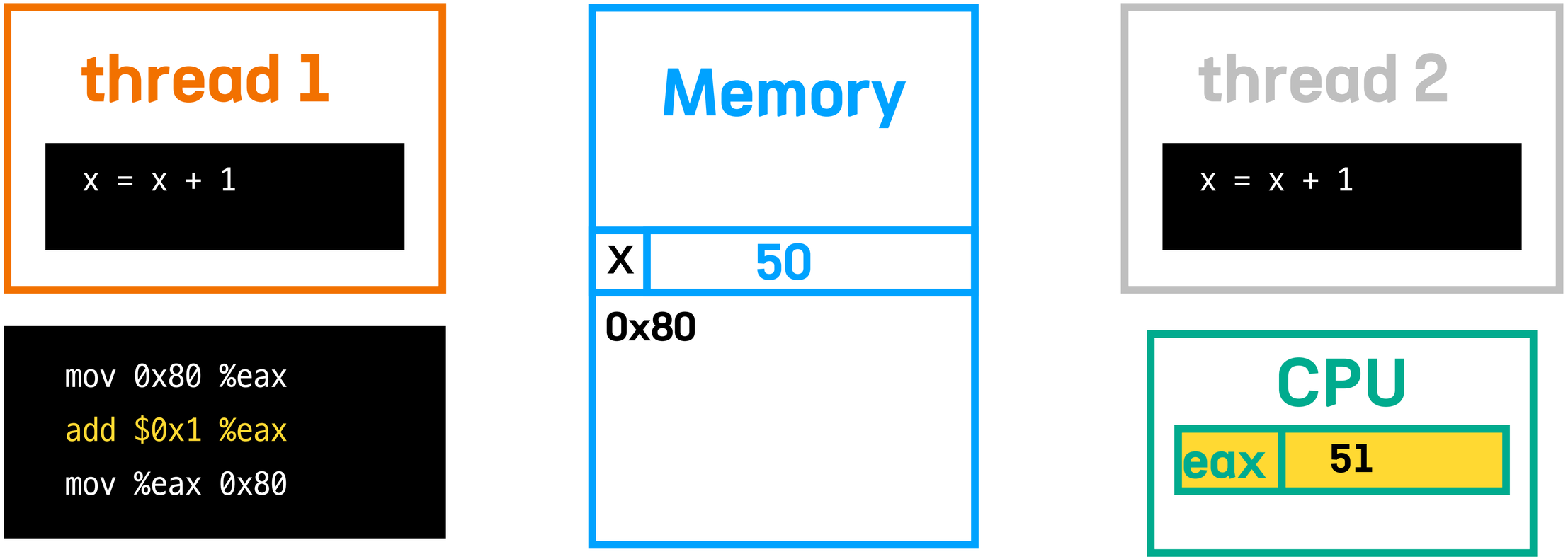
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

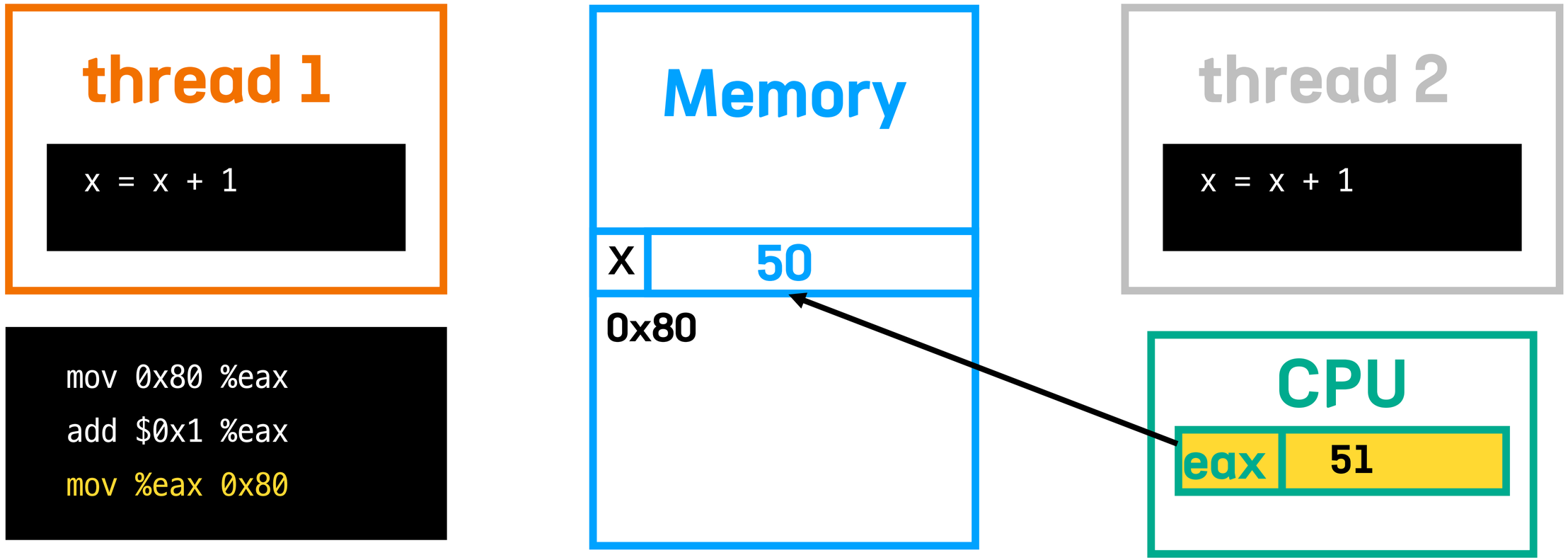
같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

thread와 공유 변수

같은 메모리 공간을 공유하면서 문제가 발생하는 경우: 동일한 변수에 접근



process와 thread

임계 영역과 상호 배제

경쟁 조건 (race condition)
명령어의 실행 순서에 따라 결과가 달라지는 상황

임계 영역 (critical section)
race condition을 유발하는 코드 부분

```
x = x + 1
```

상호 배제 (mutual exclusion)
임계 영역 내 코드를 수행하는 동안
다른 thread가 실행할 수 없게끔 보장해 주는 것

```
mov 0x80 %eax
add $0x1 %eax
mov %eax 0x80
```

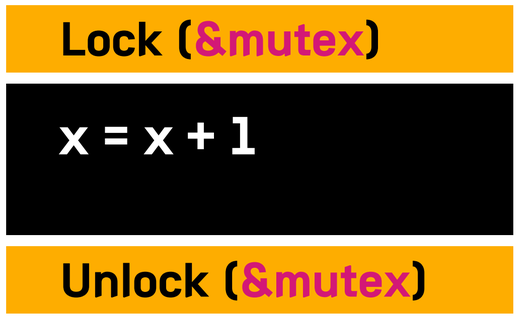
세 개의 코드가 모두 원자적으로 실행되길 바람

세 개의 코드가 하나의 작은 단위처럼,
모두 수행되거나 / 수행되지 않거나의 결과를 가지는 것

thread lock

lock의 개념 및 평가

1. 현재 lock 잡은 thread 없음!
Thread 1이 lock 잡고 임계 영역에 들어감



thread 1

2. 임계 영역에 들어가고자 lock을 잡으려 하였으나
이미 다른 쓰레드가 lock 잡고 수행 중임

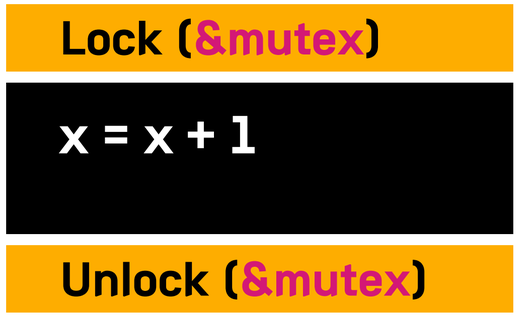
thread 2

3. Thread 1이 작업을 다 수행한 후 unlock
이제 대기중이던 thread 2가 lock을 잡을 수 있게 됨

& mutex: lock 변수로,
현재 이 자원이 잠겨 있는지열려 있는지를 기록한다.
0 = 자원 사용 가능(unlock), 1 = 자원 사용 중(unlock)

thread lock

lock의 개념 및 평가



lock은 어떻게 구현해야 할까? 어떻게 만드는 것이 좋은 lock일까?

1. 상호 배제를 제대로 지원하는가
2. thread들이 lock 획득에 대한 공정한 기회를 받는가
3. 성능 평가
   - thread가 lock을 잡는 데 발생하는 비용은 얼마나 되는가?
     (lock 사용 시 발생하는 overhead)
   - 여러 thread가 단일 CPU에서 lock을 획득하려고 경쟁하는 경우

thread lock

lock의 구현: interrupt 제어

가장 간단한 생각: 임계 구역 수행 중일 때 CPU가 다른 작업을 하지 못하게 막는다

interrupt: CPU가 하던 일을 잠깐 멈추고,
갑자기 들어온 외부 신호(키보드, 마우스, 알림 등)를 먼저 처리하게 만드는 장치

```
Void lock( ){
DisableInterrupts( );
}
```

```
Void unlock( ){
  EnableInterrupts( );
}
```

한계: CPU가 여러 개인 경우 제어가 어려움 / 중요한 인터럽트마저 무시하는 경우 발생

thread lock

lock의 구현: 하나의 flag 사용

하드웨어를 사용하지 않고 변수만으로 lock을 구현할 수는 없을까?

```
Void lock(lock_t *mutex ){
  while(mutex → flag == 1)
   ;
  mutex → flag = 1;
}
```

lock_t
flag = 0

```
Void unlock(lock_t *mutex ){
   mutex → flag = 0;
}
```

thread lock

lock의 구현: 하나의 flag 사용

문제점! 상호 배제가 안정적으로 보장되지 못함

```
Void lock(lock_t *mutex ){
  while(mutex → flag == 1)
   ;
  mutex → flag = 1;
}
```

lock_t

flag = 0

```
Void unlock(lock_t *mutex ){
    mutex → flag = 0;
}
```

thread 1

thread 2

thread lock

lock의 구현: 하나의 flag 사용

문제점! 상호 배제가 안정적으로 보장되지 못함

```
Void lock(lock_t *mutex ){
  while(mutex → flag == 1)
   ;
  mutex → flag = 1;
}
```

lock_t
flag = 1

```
Void unlock(lock_t *mutex ){
   mutex → flag = 0;
}
```

thread 1

thread 2

임계 영역으로 들어옴

thread lock

lock의 구현: 하나의 flag 사용

문제점! 상호 배제가 안정적으로 보장되지 못함

```
Void lock(lock_t *mutex ){
  while(mutex → flag == 1)
   ;
  mutex → flag = 1;
}
```

lock_t
flag = 1

```
Void unlock(lock_t *mutex ){
   mutex → flag = 0;
}
```

thread 1

임계 영역으로 들어옴

thread 2

임계 영역으로 들어옴

thread lock

lock의 구현: peterson의 알고리즘

하나의 flag만으로는 상호 배제를 완벽하게 수행할 수 없다!
또 다른 변수를 하나 더 추가해서 조정하자

```
Void lock(lock_t *mutex ){
  flag[self] = 1;
  turn = 1 - self
  while(flag[1 - self] == 1
  && turn == 1 - self)
   ;
}
```

```
flag[2] = 0
turn = 0
```

```
Void unlock(lock_t *mutex ){
   flag[self] = 0;
}
```

2개의 thread만 안정적으로 처리할 수 있음
최근의 하드웨어에서는 명령어가 순차적으로 실행된다는 보장이 없어서, 사용 불가
→ **하드웨어 명령어를 통해 수행하자!**

thread lock

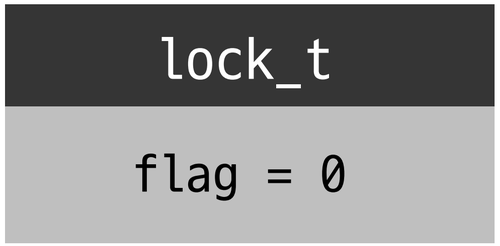
lock의 구현: Test-And-Set을 사용한 spin lock

```
Int TestAndSet (int *old_ptr, int new){
  Int old = *old_ptr;
  *old_ptr = new;
  return old
}
```

CPU의 하드웨어 지원 기능
: 원자적 교체 명령어

이전 값을 검사하며 동시에
새로운 값을 설정하는 것이
원자적으로 처리된다

```
Void lock(lock_t *lock){
while(TestAndSet(&lock→flag, 1) == 1);
  lock → flag = 1;
}
```



thread lock

lock의 구현: Compare-And-Swap을 사용한 spin lock

```
Int CompareAndSwap (int *ptr, int expected, int new){
  Int original = *ptr;
  if(original == expected) *ptr = new;
  return original
}
```

Test-And-Set 과의 차이점?

TAS는 무조건적으로 값을 업데이트 하기 때문에, 주로 “누가 lock을 잡았는지 " 에 사용

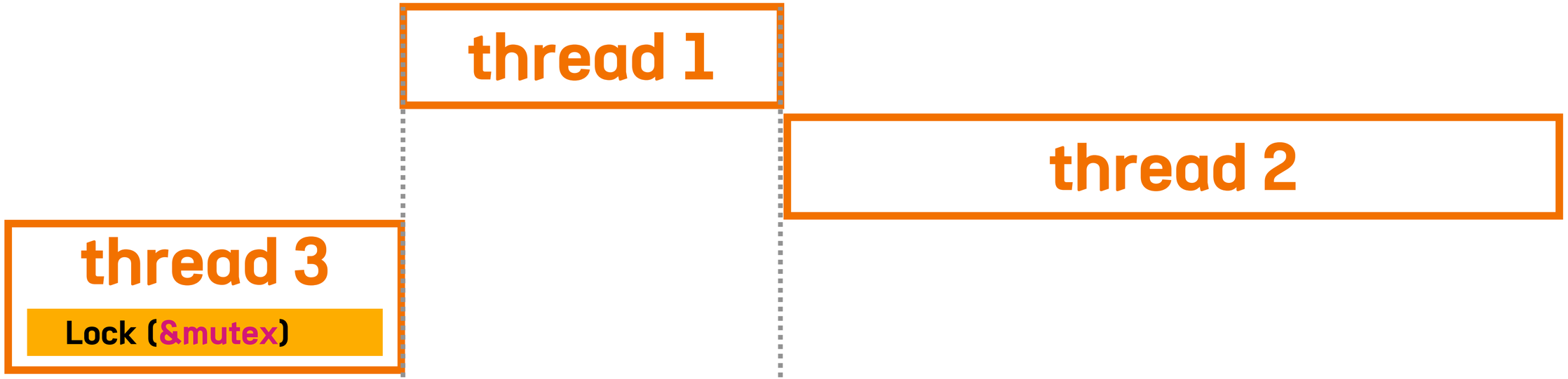
CAS는 메모리 값을 예상한 값(lock 획득 가능한 상태인지)과 비교하고 업데이트
→ 조건부 갱신, 더 범용적으로 사용된다

thread lock

spin lock의 성능

lock을 잡지 못한 thread는 계속해서 spin lock을 통해 while 문을 빙글빙글 돌 것이다
해당 thread가 CPU를 할당받아도 의미 없이 돌기만 할 것!

우선순위 역전이 발생할 수 있음: 더 먼저 수행되어야 하는 thread가 빠르게 수행되지 못한다



thread lock

spin lock의 성능

상호 배제는 제대로 지원할 수 있지만, 모든 thread가 공정한 기회를 받지 못한다.

운이 나쁜 경우, while문을 계속하여 순회한 thread가
또 다시 밀려서 계속 의미 없는 반복을 진행하고 있을 수도 있다.

성능 또한 좋지 못하다.

n개의 thread가 하나의 lock을 위해 대기하고 있다고 할 때,
CPU를 할당 받아도 의미 있는 일을 하지 못한다.

**다만, CPU가 여러 개인 경우, thread가 기다리는 시간이 줄어들기 때문에
lock을 잡기 위해 낭비하는 비용이 많지 않다!**

thread lock

lock의 구현: Load-Linked, Store-Conditional

```
Int StoreCondition (int *ptr, int value){
  if(no update to *ptr){
     *ptr = value;
     return 1;
  } else return 0;
}
```

임계 영역 진입을 위한 명령어 쌍

동일한 주소에 다른 store 값이 없는 경우에만 저장 성공

저장 성공 → return 1
저장 실패 → return 0

```
Int LoadLinked (int *ptr){
  return *ptr;
}
```

저장이 성공한다면 LoadLinked가 탑재했던 답을 갱신한다

thread lock

lock의 구현: Load-Linked, Store-Conditional

```
Void lock(lock_t *lock){
  while(1){
    while(LoadLinked(&lock → flag) == 1)
    ;
  }
  if(StoreConditional(&lock → flag, 1) == 1)
    return;
}
```

lock 함수를 어떻게 구성할까?

먼저 LoadLinked 함수로
lock의 flag를 살피고,
1인 경우 돌면서 대기한다

StoreConditional 함수를 통해
flag를 1로 업데이트한다

저장 성공 → return 1
저장 실패 → return 0

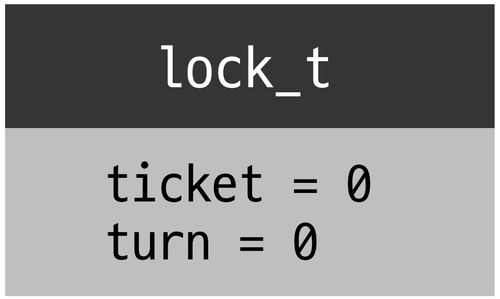
저장 성공했다면 완료(return)
실패했다면 처음부터 다시 시도

thread lock

lock의 구현: Fetch-And-Add을 사용한 ticket lock

```
Int FetchAndAdd (int *ptr){
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

ticket과 turn (차례) 조합으로 lock을 구성한다



FetchAndAdd함수로
thread의 차례를 반환 받는다

```
Void lock(lock_t *lock){
  int myturn = FetchAndAdd(&lock → ticket);
  while(lock → turn != myturn)
  ;
}
```

번호를 할당 받은 순서대로
thread가 lock 획득하도록