# Lock-based Concurrent Data Structures

## Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

# Lock-based Concurrent Data structure

- Adding locks to a data structure makes the structure **thread safe**.
  - How locks are added determine both the correctness and performance of the data structure.

# Example: Concurrent Counters without Locks

- Simple but not scalable

```
1       typedef struct __counter_t {
2               int value;
3       } counter_t;
4
5       void init(counter_t *c) {
6               c->value = 0;
7       }
8
9       void increment(counter_t *c) {
10              c->value++;
11      }
12
13      void decrement(counter_t *c) {
14              c->value--;
15      }
16
17      int get(counter_t *c) {
18              return c->value;
19      }
```

# Example: Concurrent Counters with Locks

- Add a **single lock**.
  - The lock is acquired when calling a routine that manipulates the data structure.

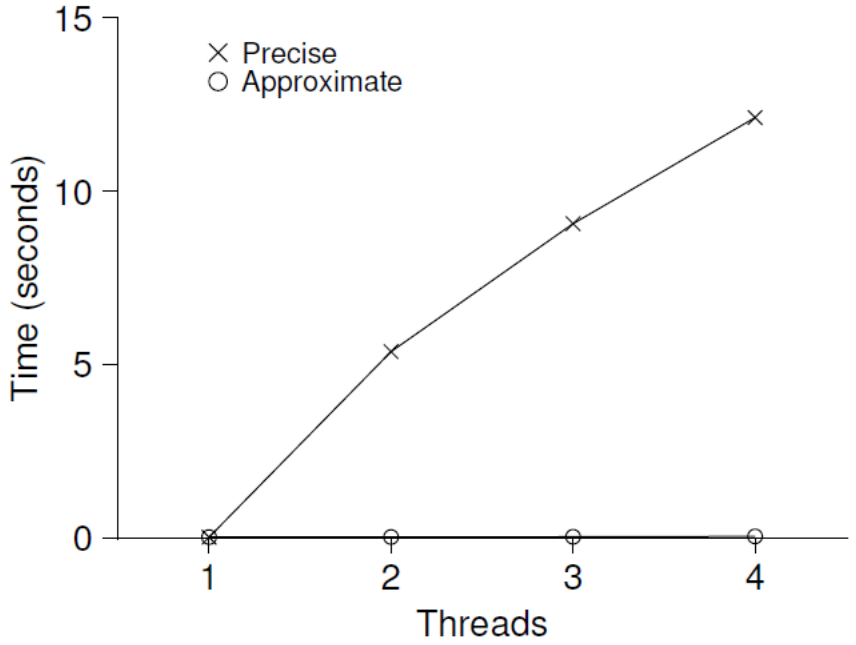
```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c->value = 0;
8               Pthread_mutex_init(&c->lock, NULL);
9       }
10
11      void increment(counter_t *c) {
12              Pthread_mutex_lock(&c->lock);
13              c->value++;
14              Pthread_mutex_unlock(&c->lock);
15      }
16
```

# Example: Concurrent Counters with Locks (Cont.)

```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```

# The performance costs of the simple approach

- Each thread updates a single shared counter.
  - Each thread updates the counter one million times (1,000,000번).
  - iMac with four Intel 2.7GHz i5 CPUs.




**Thread의 수가 증가할 수록,**
**성능이 매우 나빠짐**

**Context switching 오버헤드 누적**
(Lock에서 기다리는 thread는
blocked state로)

# Perfect Scaling (완전 확장성)

- Even though more work is done, it is **done in parallel**.
- The time taken to complete the task is *not increased*.

- **Perfect Scaling**: 다시 말하면, Thread나 CPU의 수를 N배로 늘렸다면, 처리 성능이 정확히 N배 향상되어야 함
- Perfect scaling 예시
  - 1개 thread가 작업을 10초에 끝냄
  - 2개 thread는 작업을 5초에 끝냄
  - 4개 thread는 작업을 2.5초에 끝냄

# Approximate counter

- The approximate counter works by representing …
  - A single **logical counter** via **numerous local physical counters**, on per CPU core
  - A single **logical counter** = A single **global counter**
  - There are **lock**s:
    - One for each local counter and one for the global counter

- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

# The basic idea of Approximate counting

- When a thread running on a core wishes to increment the counter.
  - It increment **its local counter**
  - Each CPU has its own local counter:
    - Threads across CPUs can update local counters *without contention*
    - Thus counter updates are scalable.
  - The local values are periodically transferred to the global counter
    - Acquire the global lock
    - Increment it by the local counter's value
    - The local counter is then reset to zero.

# The basic idea of Approximate counting (Cont.)

- How often the local-to-global transfer occurs is determined by a threshold, $S$ (Approximation Factor).
  - The smaller $S$:
    - The more the counter behaves like the *non-scalable counter*.
  - The bigger $S$:
    - The more scalable the counter.
    - The further off the global value might be from the *actual count* (actual count와 global value의 차이가 커질 수 있음)

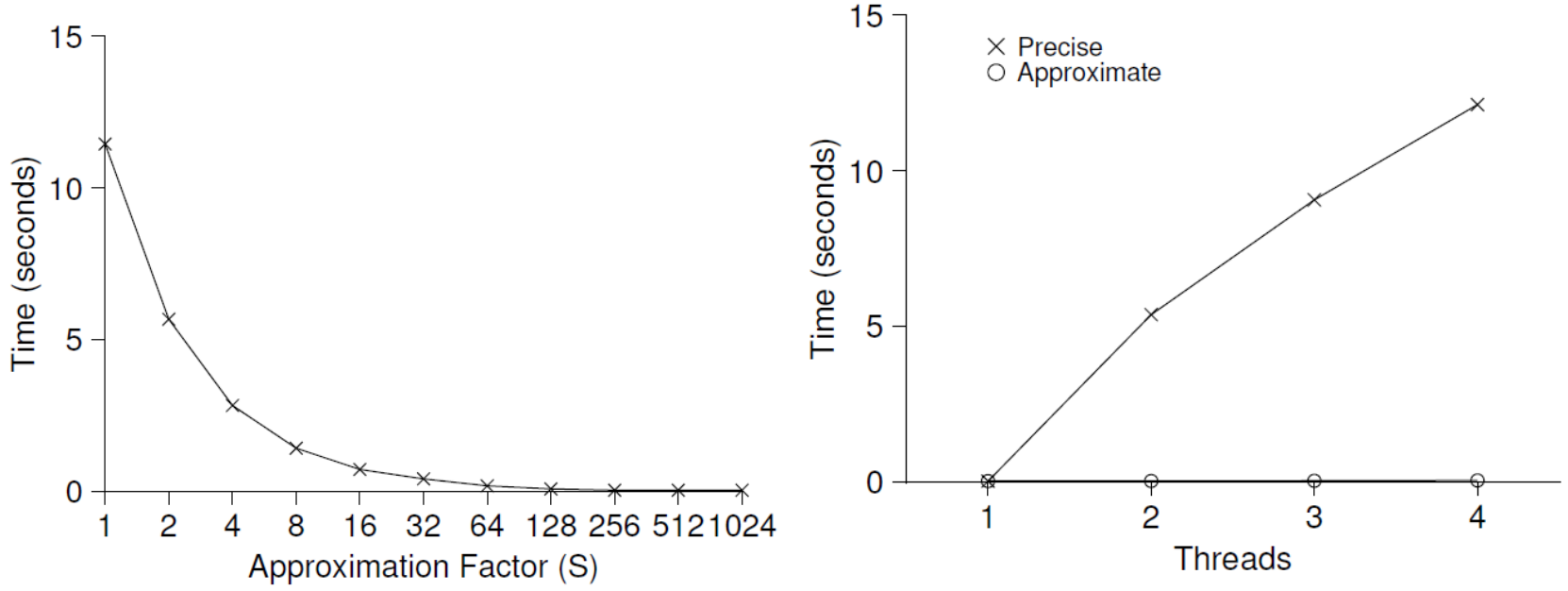
# Approximate counter example

- Tracing the Approximate Counters
  - The threshold S is set to 5.
  - There are threads on each of four CPUs
  - Each thread updates their local counters $L_1 \ldots L_4$.

| Time | $L_1$ | $L_2$ | $L_3$ | $L_4$ | G | Actual Count |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 2 |
| 2 | 1 | 0 | 2 | 1 | 0 | 4 |
| 3 | 2 | 0 | 3 | 1 | 0 | 6 |
| 4 | 3 | 0 | 3 | 2 | 0 | 8 |
| 5 | 4 | 1 | 3 | 3 | 0 | 11 |
| 6 | 5 → 0 | 1 | 3 | 4 | 5 (from $L_1$) | 13 |
| 7 | 0 | 2 | 4 | 5 → 0 | 10 (from $L_4$) | 16 |

# Importance of the threshold value S

- Each of four threads increments a counter 1 million times (1,000,000) on four CPUs.
  - Low S → Performance is **poor**, The global count is always quite **accurate**.
  - High S → Performance is **excellent**, The global count **lags (지연)**.




**Scaling Approximate Counters (Right: `S:1024`)**

# Approximate Counter Implementation

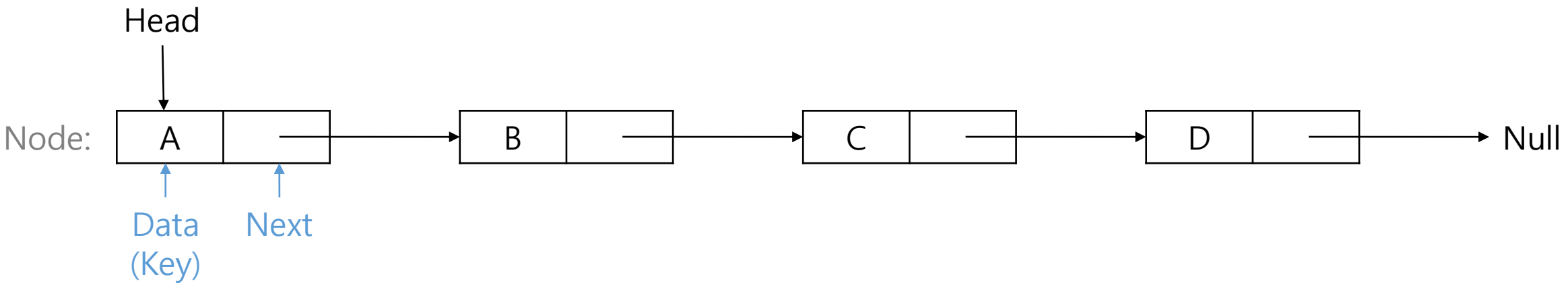
```
1       typedef struct __counter_t {
2           int global;             // global count
3           pthread_mutex_t glock;  // global lock
4           int local[NUMCPUS];     // local count (per cpu)
5           pthread_mutex_t llock[NUMCPUS]; // ... and locks
6           int threshold;          // update frequency
7       } counter_t;
8
9       // init: record threshold, init locks, init values
10      //       of all local counts and global count
11      void init(counter_t *c, int threshold) {
12          c->thres hold = threshold;
13
14          c->global = 0;
15          pthread_mutex_init(&c->glock, NULL);
16
17          int i;
18          for (i = 0; i < NUMCPUS; i++) {
19              c->local[i] = 0;
20              pthread_mutex_init(&c->llock[i], NULL);
21          }
22      }
23
```

# Approximate Counter Implementation (Cont.)

```
(Cont.)
24      // update: usually, just grab local lock and update local amount
25      //         once local count has risen by 'threshold', grab global
26      //         lock and transfer local values to it
27      void update(counter_t *c, int threadID, int amt) {
28          pthread_mutex_lock(&c->llock[threadID]);
29          c->local[threadID] += amt;      // assumes amt > 0
30          if (c->local[threadID] >= c->threshold) { // transfer to global
31              pthread_mutex_lock(&c->glock);
32              c->global += c->local[threadID];
33              pthread_mutex_unlock(&c->glock);
34              c->local[threadID] = 0;
35          }
36          pthread_mutex_unlock(&c->llock[threadID]);
37      }
38
39      // get: just return global amount (which may not be perfect)
40      int get(counter_t *c) {
41          pthread_mutex_lock(&c->glock);
42          int val = c->global;
43          pthread_mutex_unlock(&c->glock);
44          return val;    // only approximate!
45      }
```

# Concurrent Linked Lists

- Linked list: 각 노드들이 next pointer로 이어진 구조
  - 만약, thread가 linked list를 동시에 접근하려고 한다면?
  - **Concurrent Linked Lists**:
    - List 전체에 lock을 건다 (simple way)
    - 언제? 각 thread 가 concurrent linked list 에 접근하려고 할 때 (e.g., insert, search …)



# Concurrent Linked Lists

```
1       // basic node structure
2       typedef struct __node_t {
3               int key;
4               struct __node_t *next;
5       } node_t;
6
7       // basic list structure (one used per list)
8       typedef struct __list_t {
9               node_t *head;
10              pthread_mutex_t lock;
11      } list_t;
12
13      void List_Init(list_t *L) {
14              L->head = NULL;
15              pthread_mutex_init(&L->lock, NULL);
16      }
```

# Concu

```
18      int List_Insert(list_t *L, int key) {
19              pthread_mutex_lock(&L->lock);
20              node_t *new = malloc(sizeof(node_t));
21              if (new == NULL) {
22                      perror("malloc"); // stderr로 오류 출력
23                      pthread_mutex_unlock(&L->lock);
24                      return -1; // fail
25              }
26              new->key = key;
27              new->next = L->head;
28              L->head = new;
29              pthread_mutex_unlock(&L->lock);
30              return 0; // success
31      }
32
33      int List_Lookup(list_t *L, int key) {
34              pthread_mutex_lock(&L->lock);
35              node_t *curr = L->head;
36              while (curr) {
37                      if (curr->key == key) {
38                              pthread_mutex_unlock(&L->lock);
39                              return 0; // success
40                      }
41                      curr = curr->next;
42              }
43              pthread_mutex_unlock(&L->lock);
44              return -1; // failure
45      }
```

# Concurrent Linked Lists (Cont.)

- The code **acquires** a lock in the insert routine upon entry
- The code **releases** the lock upon exit
  - If `malloc()` happens to *fail*, the code must also release the lock before failing the insert.
  - This kind of exceptional control flow has been shown to be quite error-prone (오류가 발생하기 쉽다 → Deadlock)
    - Malloc 실패 시, 락을 해제하지 않고 return 하면 deadlock
    - While 안에서, 락을 해제하지 않고 return 하면 deadlock
  - **Solution**: The lock and release *only surround* the actual critical section in the insert code

# Concurrent Linked List: Rewritten

```
1       void List_Init(list_t *L) {
2               L->head = NULL;
3               pthread_mutex_init(&L->lock, NULL);
4       }
5
6       void List_Insert(list_t *L, int key) {
7               // synchronization not needed
8               node_t *new = malloc(sizeof(node_t));
9               if (new == NULL) {
10                      perror("malloc"); // stderr로 오류 출력
11                      return;
12              }
13              new->key = key;
14
15              // just lock critical section
16              pthread_mutex_lock(&L->lock);
17              new->next = L->head;
18              L->head = new;
19              pthread_mutex_unlock(&L->lock);
20      }
21
```

# Concurrent Linked List: Rewritten (Cont.)

```
(Cont.)
22      int List_Lookup(list_t *L, int key) {
23              int rv = -1;
24              pthread_mutex_lock(&L->lock);
25              node_t *curr = L->head;
26              while (curr) {
27                      if (curr->key == key) {
28                              rv = 0;
29                              break;
30                      }
31                      curr = curr->next;
32              }
33              pthread_mutex_unlock(&L->lock);
34              return rv; // now both success and failure
35      }
```

# Scaling Linked List

- Problems of Concurrent Linked List:
  - **Bottleneck**: lock 대기시간 증가, context switching overhead 증가
- Hand-over-hand locking (lock coupling)
  - Add **a lock per node** of the list instead of having a single lock for the entire list.
  - When traversing the list,
    - First grabs the next node's lock.
    - And then releases the current node's lock.
  - Enable a high degree of concurrency in list operations.
    - **However**, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*
      *(실제로 list를 탐색할 때, 각 node의 lock을 잠금/해제의 오버헤드가 엄청남)*
      *(코드 복잡성 증가 → Deadlock 발생 가능성 증가)*

# Scaling Linked List

- Hand-over-hand locking (lock coupling)

```
33      int List_Lookup(list_t *L, int key) {
34              node_t *curr = L->head;
35              while (curr) {
36                      pthread_mutex_lock(&L->lock);
37                      if (curr->key == key) {
38                              pthread_mutex_unlock(&L->lock);
39                              return 0; // success
40                      }
41                      curr = curr->next;
42                      pthread_mutex_unlock(&L->lock);
43              }
44              return -1; // failure
45      }
```

# Michael and Scott Concurrent Queues

- There are two locks
  - One for the **head** of the queue.
  - One for the **tail**.
  - The goal of these two locks is to enable concurrency of ***enqueue*** and ***dequeue*** operations.

- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations

# Concurrent Queues (Cont.)

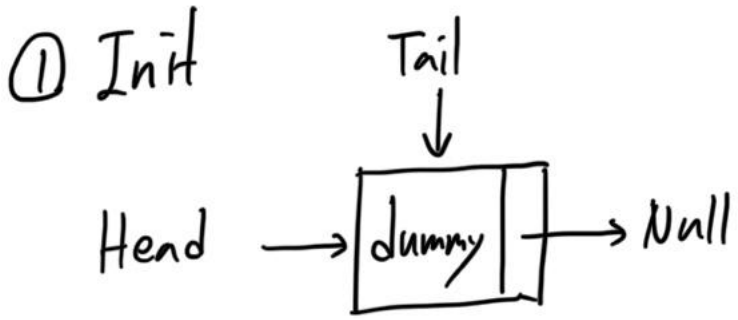
```
1       typedef struct __node_t {
2               int value;
3               struct __node_t *next;
4       } node_t;
5
6       typedef struct __queue_t {
7               node_t *head;
8               node_t *tail;
9               pthread_mutex_t headLock;
10              pthread_mutex_t tailLock;
11      } queue_t;
12
13      void Queue_Init(queue_t *q) {
14              node_t *tmp = malloc(sizeof(node_t));
15              tmp->next = NULL;
16              q->head = q->tail = tmp;
17              pthread_mutex_init(&q->headLock, NULL);
18              pthread_mutex_init(&q->tailLock, NULL);
19      }
20
```

# Concurrent Queues (Cont.)

```
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24              tmp->value = value;
25              tmp->next = NULL;
26
27              pthread_mutex_lock(&q->tailLock);
28              q->tail->next = tmp;
29              q->tail = tmp;
30              pthread_mutex_unlock(&q->tailLock);
31      }
32
33      int Queue_Dequeue(queue_t *q, int *value) {
34              pthread_mutex_lock(&q->headLock);
35              node_t *tmp = q->head;
36              node_t *newHead = tmp->next;
37              if (newHead == NULL) {
38                      pthread_mutex_unlock(&q->headLock);
39                      return -1; // queue was empty
40              }
41              *value = newHead->value;
42              q->head = newHead;
43              pthread_mutex_unlock(&q->headLock);
44              free(tmp);
45              return 0;
46      }
```

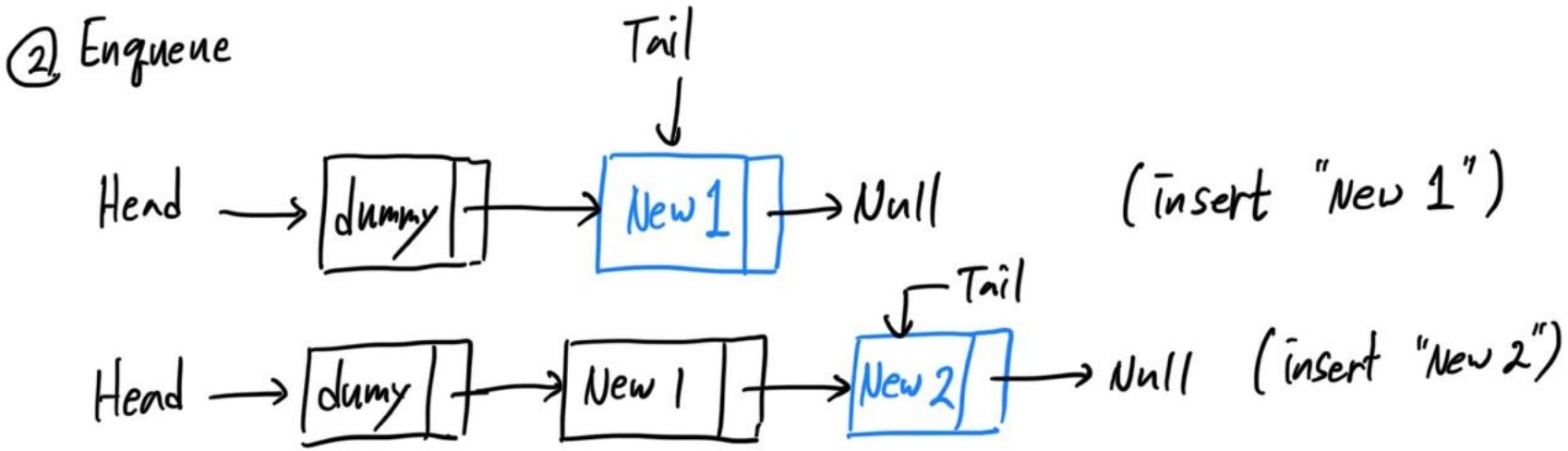
# Concurrent Queues (Cont.)

```
13      void Queue_Init(queue_t *q) {
14              node_t *tmp = malloc(sizeof(node_t));
15              tmp->next = NULL;
16              q->head = q->tail = tmp;
17              pthread_mutex_init(&q->headLock, NULL);
18              pthread_mutex_init(&q->tailLock, NULL);
19      }
20
```



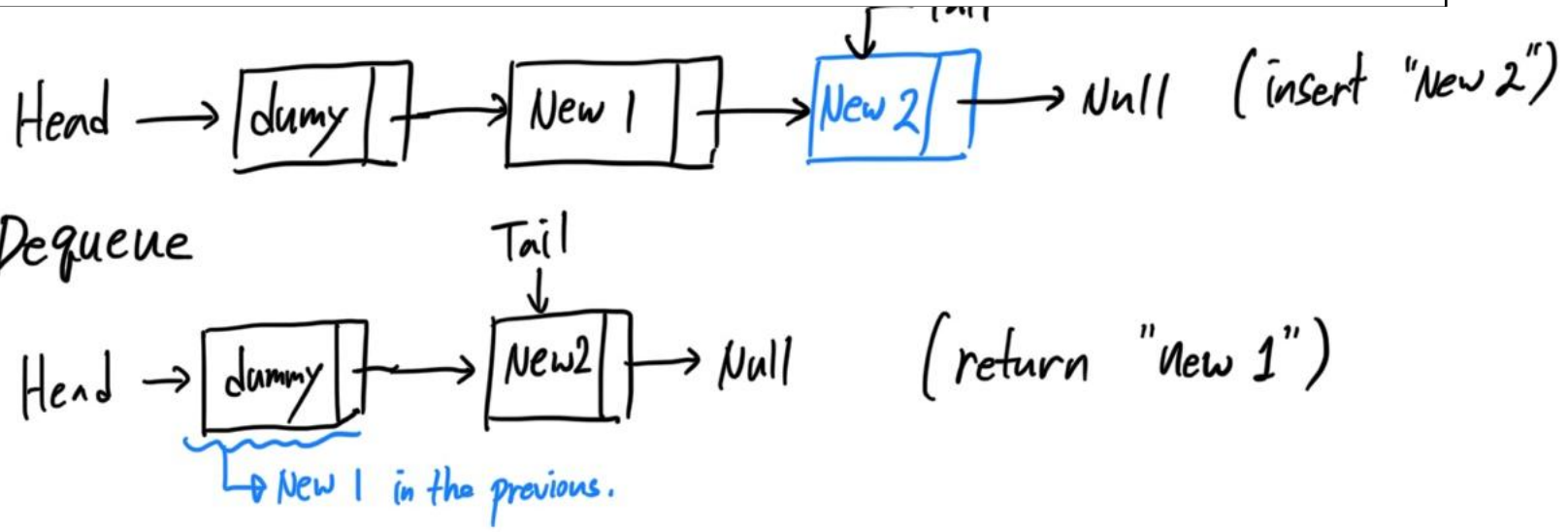
# Concurrent Queues (Cont.)

```
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24              tmp->value = value;
25              tmp->next = NULL;
26
27              pthread_mutex_lock(&q->tailLock);
28              q->tail->next = tmp;
29              q->tail = tmp;
30              pthread_mutex_unlock(&q->tailLock);
31      }
```



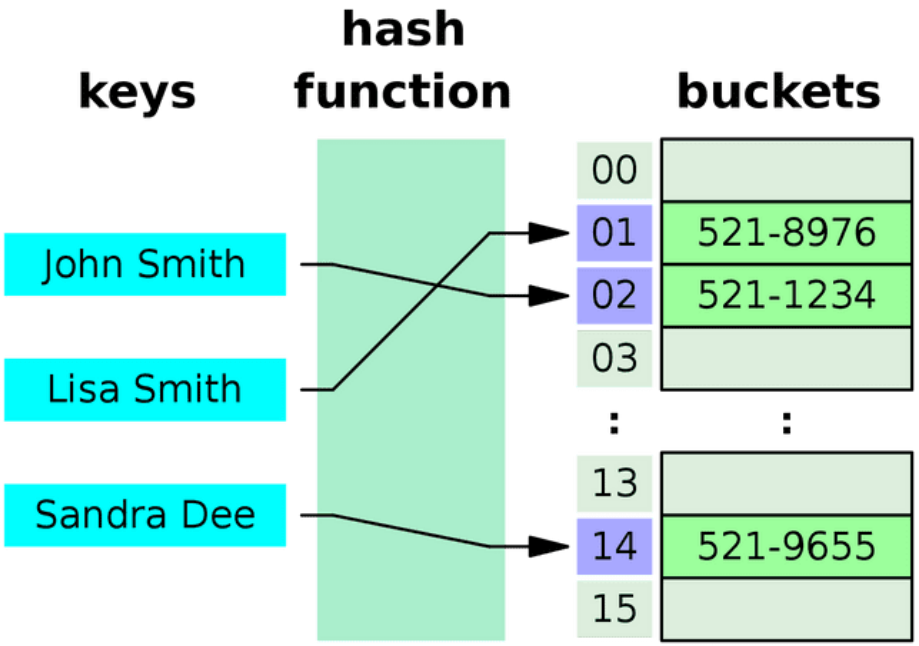
# Concurrent Queues (Cont.)

```
33      int Queue_Dequeue(queue_t *q, int *value) {
34              pthread_mutex_lock(&q->headLock);
35              node_t *tmp = q->head;
36              node_t *newHead = tmp->next;
37              if (newHead == NULL) {
38                      pthread_mutex_unlock(&q->headLock);
39                      return -1; // queue was empty
40              }
41              *value = newHead->value;
42              q->head = newHead;
43              pthread_mutex_unlock(&q->headLock);
44              free(tmp);
45              return 0;
46      }
```



# Concurrent Hash Table

- Concurrent Hash Table
  - 여러 Thread 가 동시에 같은 bucket에 접근 X
  - 전체 Hash Table을 하나의 lock을 사용하면 bottleneck 발생
  - 각 Bucket 단위로 lock 분할 (=Fine-grained locking)

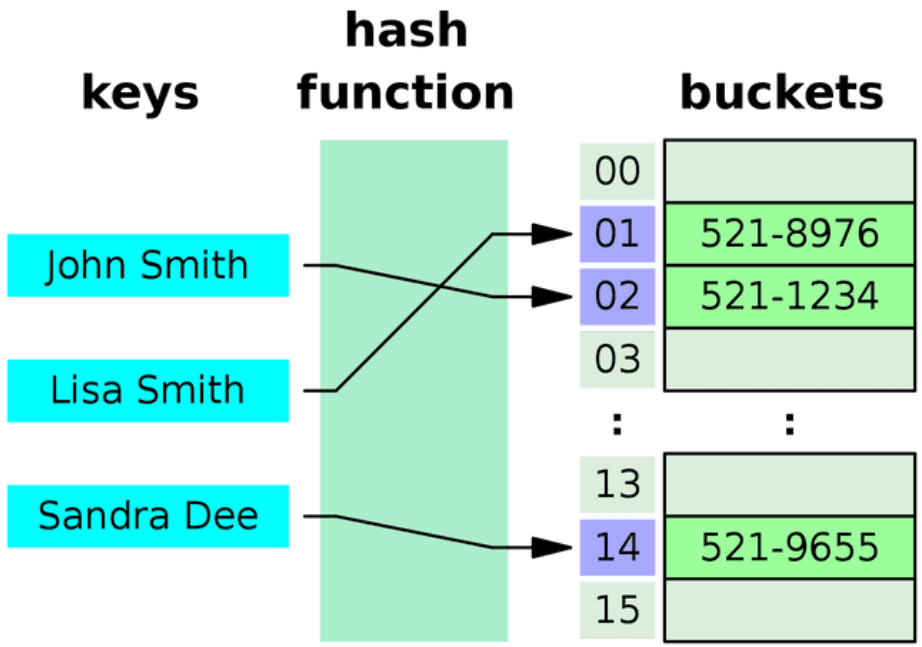



Hash table 예시

(this image is from https://renuevo.github.io/data-structure/hashtable/)

# Concurrent Hash Table

- Focus on a simple hash table
  - The hash table does not resize
  - Built using the concurrent lists
  - It uses a lock per hash bucket each of which is represented by *a list*.




Hash table 예시

(this image is from https://renuevo.github.io/data-structure/hashtable/)

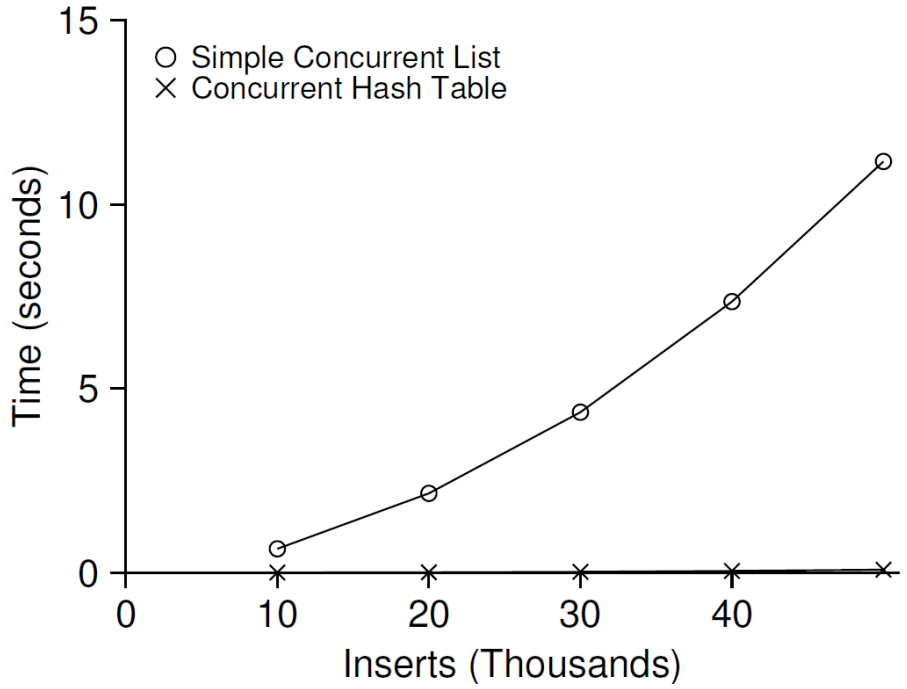
# Concurrent Hash Table (Cont.)

```
1       #define BUCKETS (101)
2
3       typedef struct __hash_t {
4               list_t lists[BUCKETS];
5       } hash_t;
6
7       void Hash_Init(hash_t *H) {
8               int i;
9               for (i = 0; i < BUCKETS; i++) {
10                      List_Init(&H->lists[i]);
11              }
12      }
13
14      int Hash_Insert(hash_t *H, int key) {
15              int bucket = key % BUCKETS;
16              return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20              int bucket = key % BUCKETS;
21              return List_Lookup(&H->lists[bucket], key);
22      }
```

List는 concurrent linked list 사용 (p19)

# Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads.
  - iMac with four Intel 2.7GHz i5 CPUs.




**The simple concurrent hash table scales magnificently**

# Summary

- Sloppy counter
- Concurrent list
- Concurrent queue
- Concurrent hash table