KYUNGPOOK NATIONAL UNIVERSITY

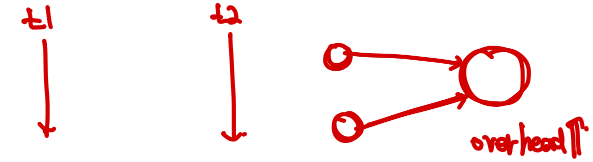
Lock-based Concurrent Data Structures

→ 자료구조를 어떻게 thread-safety 하게!

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Concurrent Counters



- **A concurrent counter requires locks to make it work correctly**
  - Before accessing the counter variable, each thread must acquire the lock

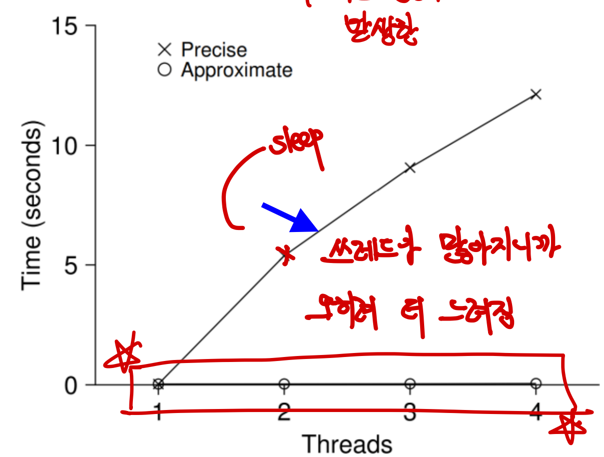
```
typedef struct __counter_t {
    int             value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

lock 초기화.

lock, unlock은 사용하는 데이터 자체에 둠

→ 쓰레드를 1개만 사용할 때가 2개를 사용할 때보다 더 빠른 현상이 발생함

- **The problem of this implementation is performance**
  - Each thread update the counter one million times
  - Varying the number of threads (1~4)
  - Running on iMac with four Intel 2.7GHz i5 CPUs
  - Unfortunately, the performance scales poorly
  - Adding thread leads to massive slowdown



Scalable Counting

Core에 따른 성능 향상이 X

↦ 쓰레드가 여러개로 성능 저하가 발생X

thread 따라

- **Perfect scaling: though more work is done, it is done in parallel**
  - The approximate counter works by representing a single logical counter via many local physical counters, one per CPU core, and a single global counter

    logical counter.
  - Each local counter and the global counter have their own lock

- **The basic idea of approximate counting is**

  thread 따라 다르고 (logical) → 나중에 다 더함 (global)

  1) Each thread running on a given core increments its local counter using lock
  2) The local values are periodically transferred to the global counter using lock
  3) The local counter resets to zero after the transfer

  J를 넘으면 local counter의 값을 global counter에 반영

- **Tracing the approximate counters:** Core 4

S = 5

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$\boxed{5 \rightarrow 0}$	1	3	4	5 (from L_1)
7	0	2	4	$\boxed{5 \rightarrow 0}$	10 (from L_4)

5 threshold

global counter를 업데이트 하도록

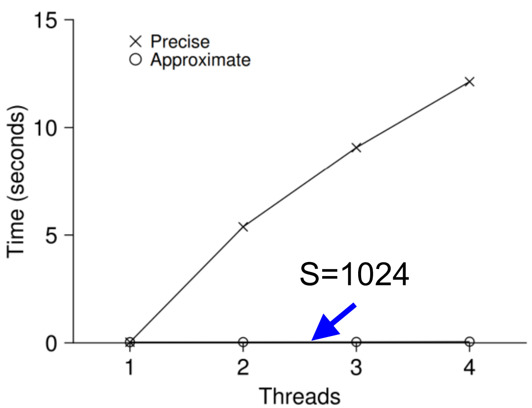
Scalability Factor

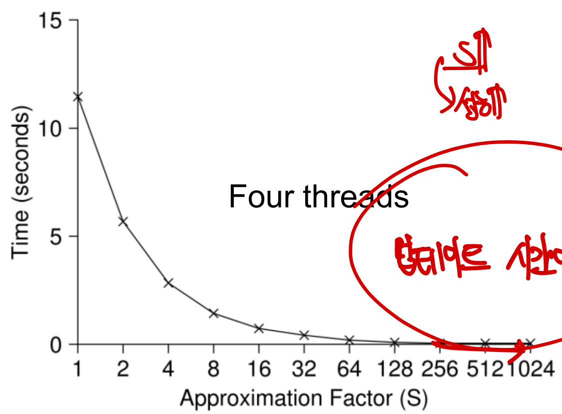
S가 몇일때가 가장 좋을까?

- **How often this local-to-global transfer occurs is determined by a threshold S**

  속도, 정확도 trade-off.

  – The smaller S is, the more the counter behaves like the non-scalable counter
  – The bigger S is, the more scalable the counter, but the further off the global value might be from the actual count
- **The performance of approximate counter is excellent**
  – The time does not increase significantly as the number of thread increases
  – If S is high, the performance is excellent but the global counter lags
  – If S is low, the performance is poor (accurate)





S↑ 성능↑

업데이트 시간↓↓

cpu 개수만큼 local counter 생성

```
typedef struct __counter_t {
    int             global;           // global count
    pthread_mutex_t glock;            // global lock
    int             local[NUMCPUS];   // per-CPU count
    pthread_mutex_t llock[NUMCPUS];   // ... and locks
    int             threshold;        // update frequency
} counter_t;

void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;
    if (c->local[cpu] >= c->threshold) {
        // transfer to global (assumes amt>0)
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}
```

Concurrent Linked Lists

- **We next examine a more complicated structure, the linked list**

```
// basic node structure
typedef struct __node_t {
    int                 key;
    struct __node_t     *next;
} node_t;

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}
```

lock은 1개인데

unlock을 위한 코드가 2개?

↓

변경.

실패할 때 mutex를 잡고 들어감.

```
// basic list structure (one used per list)
typedef struct __list_t {
    node_t              *head;
    pthread_mutex_t     lock;
} list_t;

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

끝날 때는 lock을 풀어야 함

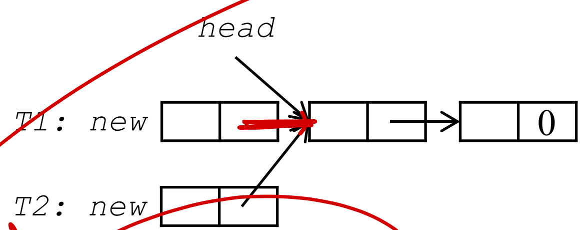
  - The code acquires a lock in the insert routine upon entry and releases upon exit
  - If **`malloc()`** fails, the code must also release the lock before failing the insert
- **This kind of exceptional control flow is to be quite error prone**
  - Can we rewrite the insert and lookup codes to remain correct under concurrent insert but avoid the case where the failure path requires the call to unlock?
  - The answer is YES

Revised Concurrent Linked List

- **We can do the lock and release only surround the actual critical section and a common exit path is used in the lookup code**
  - The former works because part of the insert actually need not be locked
  - Only when updating the shared list does a lock need to be held

엄청난 동시성을 제공하지만. "overhead ↑"

```
void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}
```



→ 삽입해 줄때만 mutex-lock을 해줘야 한다.

```
int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

통합된 창구.

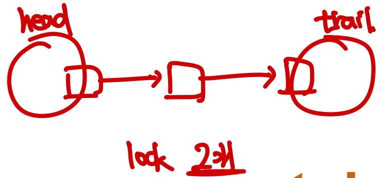
찾는 경우는 O

- **Scalable linked list can be achieved by adding a lock per node**
  - This enables a high degree of concurrency in list operations
  - However, it is hard to make it faster than the single lock approach

노드마다 lock

Concurrent Queues

→ head와 tail 각각에 lock을 주어서 동시에 pop, push를 할 수 있게.



- **There is always a standard method to make a concurrent data structure: add a big lock**
  - For concurrent queues, we can achieve scalability by adding two locks, each for head and tail of the queue, enabling concurrency of enqueue and dequeue

```c
typedef struct __node_t {
    int              value;
    struct __node_t  *next;
} node_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}

void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tail_lock);
}
```

```c
typedef struct __queue_t {
    node_t            *head;
    node_t            *tail;
    pthread_mutex_t   head_lock, tail_lock;
} queue_t;
```

↳ 앞 뒤 데이터가 이동하니 lock도 2개

```c
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    node_t *tmp = q->head;
    node_t *new_head = tmp->next;
    if (new_head == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return -1; // queue was empty
    }
    *value = new_head->value;
    q->head = new_head;
    pthread_mutex_unlock(&q->head_lock);
    free(tmp);
    return 0;
}
```

큐가 비어 있거나 데이터가 너무 많은 경우.

thread가 대기할 수 있도록하는 "bounded queue"

Concurrent Hash Table

- **We finally consider a simple and widely applicable concurrent data structure, the hash table** "제일 중요함"
  - Let's focus on a simple hash table that does not resize
  - This concurrent hash table is straightforward, is built using the concurrent lists we developed earlier
- **The performance of this hash table is good**
  - Instead of a single lock for the entire structure, it uses a lock per hash bucket "bucket이 다른 경우만!"

bucket이 다르면 동시에 가능.

linked list의 함수 그대로 사용

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```