Counters
Linked Lists
Queues
Hash Tables

16. Lock-based Concurrent Data Structures

여러 개의 thread가 동시에 접근하는 data structures

▼ 어떻게 lock을 추가? +) lock을 어떻게 하면 효율적으로 사용할 수 있을 지에 대해서도 알아보자!

- Correctness : 어떻게 lock을 적절하게 add?
- Concurrency : 어떻게 하면 높은 성능으로 lock을 추가
  - 동시에 여러 thread가 각 자료구조에 접근할 때 어떻게?

▼ Counter

mutex 사용해서 counter synchronization

- Concurrent Counters

```
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);
}

//counter 증감하는 구간 -> critical section으로 구현
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;
}
```

critical section
critical section
critical section

→ but, core 개수 늘어나면 병렬성 떨어짐 ⇒ local/global counter

- Scalable Counting
  - Sloppy counter ~> 대중?.. 헐렁한 counter
    - Logical counter
      - CPU core마다 각각 존재
      - global counter
      - Locks → 각 local counter마다, global counter마다 존재
    - basic idea
      ① local counter에 대한 증감은 다른 thread의 영향 없이 증감
      ② 주기적으로 local counter의 값을 global counter에 반영할 수 있도록 설정
        - global lock(①)을 얻고 local counter value(②)에 의해 증가
        - local counter → 0으로 reset (update 후에)
      ③ local-global 값 전달을 얼마나 자주 할 지 결정해야 함
    - example ⇒ local counter가 n개 되면 global에 반영하는 example

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	5→0	1	3	4	5
7	0	2	4	5→0	10

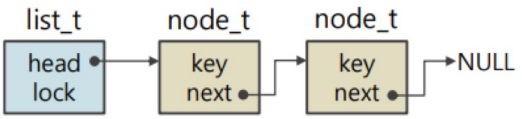
반영 후 0이됨

반영후 0이됨

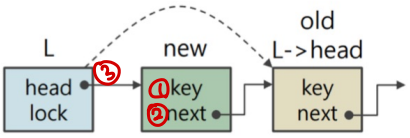
*sloppy counter

```c
typedef struct __counter_t {
  int global;                 // global lock 여부
  pthread_mutex_t glock;      // global lock
  int local[NUMCPUS];         // local counter
  pthread_mutex_t llock[NUMCPUS]; // local lock
  int threshold; // update frequency -> local-global 값 전달
} counter_t;

void init(counter_t *c, int threshold) {
  c->threshold = threshold;
  c->global = 0;
  pthread_mutex_init(&c->glock, NULL);
  int i;
  for (i = 0; i < NUMCPUS; i++) {
    c->local[i] = 0;
    pthread_mutex_init(&c->llock[i], NULL);
  }
}

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

int get(counter_t *c) {
  pthread_mutex_lock(&c->glock); // global lock
  int val = c->global;
  pthread_mutex_unlock(&c->glock);
  return val; // only approximate!
}
```

▼ Concurrent Linked Lists




```
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
  pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(&L->lock, NULL);
}
```




```
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
}
```

```
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
}
```

여러 thread들이 새로운 node 삽입할 때 lock → but, 성능 저하, 꼭 필요할 때만 lock..

- rewritten ver ⇒ 개선된 ver

① insert → 병렬성 개선

```
void List_Insert(list_t *L, int key) {
  // synchronization not needed → 불필요한 lock을 제거 ⇒ 병렬성 ↑
  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);
}
```

critical section → 실제 필요한 critical section 부분

② Lookup → CPU가 여러 개 일때의

```
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; → unlock 대신 break로 수정
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return rv; // bug pruning
}
```

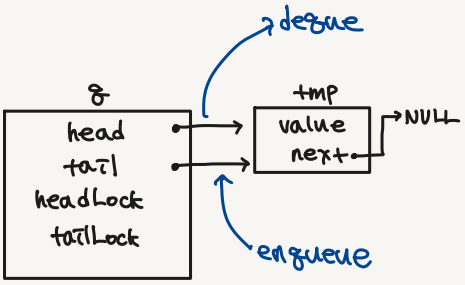
↳ lock, unlock이 대칭함
대칭할 때 어느 경우에서나 꼭 return되어야 함.

- Hand-over-hand locking
  - list의 node마다 lock 넣어야 함(전체 list에 하나의 lock이 필요한 것 x)
  - list traverse → 다음 node lock 한 뒤에 현재 node lock을 release
  - 각 node마다 lock을 acquiring 하고 releasing 하는 overhead 발생
- Non-blocking linked list
  - compare-and-swap 사용

```
void List_Insert(list_t *L, int key) {
...
RETRY: next = L->head;
       new->next = next;
       if (CAS(&L->head, next, new) == 0)
         goto RETRY;
}
```

▼ Concurrent Queues (FIFO)

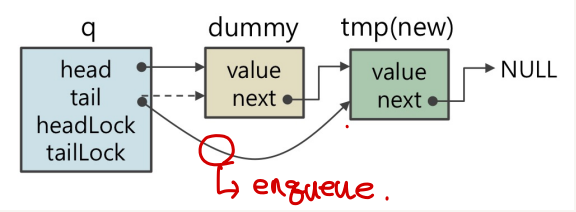
```c
typedef struct __node_t {
  int value;
  struct __node_t *next;
} node_t;

typedef struct __queue_t {
  node_t *head; // out (dequeue)
  node_t *tail; // in (enqueue)
  pthread_mutex_t headLock;
  pthread_mutex_t tailLock;
} queue_t; //head와 동일한 역할

void Queue_Init(queue_t *q) {
  node_t *tmp = malloc(sizeof(node_t)); // dummy node
  tmp->next = NULL;
  q->head = q->tail = tmp;
  pthread_mutex_init(&q->headLock, NULL);
  pthread_mutex_init(&q->tailLock, 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->tailLock);
  q->tail->next = tmp;
  q->tail = tmp;
  pthread_mutex_unlock(&q->tailLock);
}
```

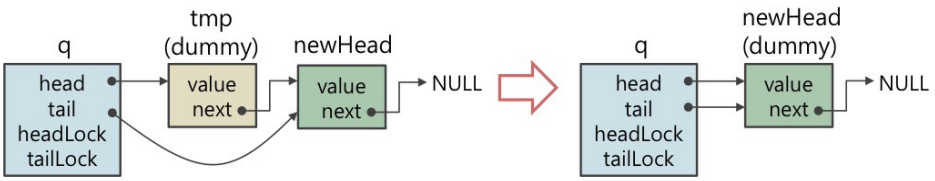


↳ 실제로 data를 담기위한 node가 아님
⇒ enqueue와 dequeue가 동일한 node에서 동시에 일어나지 않도록 만듦
(head operation과 tail operation이 동시에 일어나지 X)

→ dummy node

→ tail operation






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

head update ⇒ headLock critical section

+) enqueue, dequeue 번갈아서 사용된다면 → 중복되는 구간 없는지 확인해볼 것!

enqueue, dequeue mutex 따로 선언 → dummy node로 동시에 lock 사용 가능

▼ Concurrent Hash Table

↳ 여러 thread가 동시에 접근할 때 어떻게?

```c
#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) {
  int bucket = key % BUCKETS;
  return List_Insert(&H->lists[bucket], key);
}

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