32

락 기반의 병행 자료 구조

다른 주제로 넘어가기 전에 먼저 흔하게 사용되는 자료 구조에서 락을 사용하는 방법을 살펴보자. 자료 구조에 락을 추가하여 쓰레드가 사용할 수 있도록 만들면 그 구조는 **쓰레드 사용에 안전**(**쓰레드 안전**, **thread safe**)하다고 할 수 있다. 물론, 락이 어떤 방식으로 추가 되었느냐에 따라 그 자료 구조의 정확성과 성능을 좌우할 것이다. 주어진 도전은 다음과 같다.

핵심 질문: 자료 구조에 락을 추가하는 방법
특정 자료 구조가 주어졌을 때, 어떤 방식으로 락을 추가해야 그 자료 구조가 정확하게 동작하게 만들 수 있을까? 더 나아가, 자료 구조에 락을 추가하여 여러 쓰레드가 그 자료 구조를 동시 사용이 가능하도록 하면(**병행성**) 고성능을 얻기 위해 무엇을 해야 할까?

모든 자료 구조를 다 다루거나 병행성이 가능하도록 하는 모든 기법을 다루기는 힘들다. 수년 동안 연구되어 온 주제라서 수천 편의 논문들이 이 주제를 다루고 있기 때문이다. 여기서는 어떤 사고 방식으로 이 주제를 접근해야 하는지를 소개하고자 한다. 의문점이 생겼을 때 참고할 수 있는 좋은 자료들을 소개하고자 한다. 여기에 나온 대부분의 정보는 Moir와 Shavit의 연구를 참고하였다 [MS04].

32.1 병행 카운터

카운터는 가장 간단한 자료 구조 중 하나이다. 보편적으로 사용되는 구조이면서 인터페이스가 간단하다. 병행이 불가능한 카운터를 그림 32.1에서 정의한다.

간단하지만 확장성이 없음

보는 바와 같이, 동기화되지 않은 카운터는 몇 줄 안 되는 코드로 작성할 수 있는 평범한 자료 구조이다. 우리에게 숙제가 주어졌다. 이 코드를 어떻게 **쓰레드 안전**(**thread safe**)하게 만들 수 있을까? 그림 32.2에 그 방법을 나타내었다.

이 병행이 가능한 카운터는 간단하지만 정확하게 동작한다. 사실 이 카운터는 가장 간단하고 가장 기본적인 병행 자료 구조의 보편적인 디자인 패턴을 따른다. 자료 구조를

```
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 }
```

〈그림 32.1〉 **락이 없는 카운터**

```
1  typedef struct __counter_t {
2      int value;
3      pthread_mutex_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
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 }
```

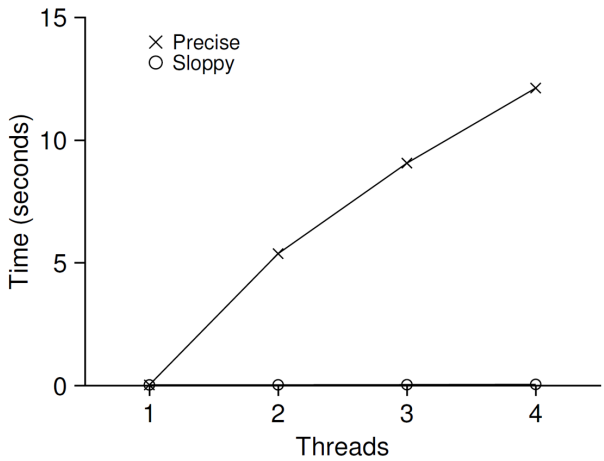
〈그림 32.2〉 **락이 있는 카운터**

조작하는 루틴을 호출할 때 락을 추가하였고, 그 호출문이 리턴될 때 락이 해제되도록 하였다. 이 방식은 **모니터**(**monitor**) [Han73]를 사용하여 만든 자료 구조와 유사하다. 모니터 기법은 객체에 대한 메소드를 호출하고 리턴할 때 자동적으로 락을 획득하고 해제한다.

이제 제대로 동작하는 병행 자료 구조를 갖게 되었다. 그렇지만 성능이 문제다. 자료 구조가 너무 느리다면 단순히 락을 추가하는 것 이상의 무엇인가를 해야 한다. 성능 최적화가 이번 장에서 다룰 내용이다. 자료 구조가 많이 느리지 않다면 할 일을 다 했다!

간단하게 해서 되는 일을 복잡하게 처리할 필요는 없다.

먼저 이 간단한 방법의 성능을 이해하기 위해서 각 쓰레드가 특정 횟수만큼 공유 카운터를 증가시키는 벤치마크를 실행하였다. 그리고 쓰레드의 개수를 증가시켜 보았다. 그림 32.3에 하나에서 네 개의 쓰레드를 사용하여 카운터를 백만 번 증가시켰을 때 총 걸린 시간을 나타내었다. 인텔 2.7 GHz i5 CPU 4개를 사용하는 iMac에서 실행하였다. 더 많은 CPU가 동작할수록 단위 시간당 총 수행량이 늘어날 것으로 기대한다.



〈그림 32.3〉 **고전적 vs. 엉성한 카운터의 성능**

범례에 precise라고 표시된 선에서 동기화된 카운터의 확장성이 떨어지는 것을 볼 수 있다. 단일 쓰레드로 카운터를 백만 번 갱신하는 데 대략 0.03초가 걸렸다. 두 개의 쓰레드로 카운트 값을 백만 번 갱신하는 데 약 5초 이상이 걸렸다.

이상적으로는 하나의 쓰레드가 하나의 CPU에서 작업을 끝내는 것처럼 멀티프로세서에서 실행되는 쓰레드들도 빠르게 작업을 처리하고 싶을 것이다. 이와 같이 동작하는 것을 **완벽한 확장성**(**perfect scaling**)이 있다. 완벽한 확장성은 더 많은 작업을 처리하더라도 각 작업이 병렬적으로 처리되어 완료 시간이 늘어나지 않는다는 것을 말한다.

확장성 있는 카운팅

수년 동안 확장성 있는 카운터를 만들기 위해 많은 연구 활동을 하였다 [MS04]. 확장성 있는 카운터는 중요한 의미를 갖는다. 최근 운영체제 성능 분석 결과에 의하면 [Boy+10], 확장 가능한 카운터가 없으면 Linux의 몇몇 작업은 멀티코어 기기에서 심각한 확장성 문제를 겪을 수 있다고 한다.

개발된 여러 기법 중 하나를 소개한다. 최근에 소개된 **엉성한 카운터**(**sloppy counter**)라고 불리는 기법이다 [Boy+10].

엉성한 카운터는 하나의 논리적 카운터로 표현되는데, CPU 코어마다 존재하는 하나의 물리적인 지역 카운터와 하나의 전역 카운터로 구성되어 있다. 어떤 기기가 네

개의 CPU를 갖고 있다면 그 시스템은 네 개의 지역 카운터와 하나의 전역 카운터를 갖고 있다. 이 카운터들 외에 지역 카운터를 위한 락들과 전역 카운터를 위한 락이 존재한다.

엉성한 카운터의 기본 개념은 다음과 같다. 쓰레드는 지역 카운터를 증가시킨다. 이 지역 카운터는 지역 락에 의해 보호된다. 각 CPU는 저마다 지역 카운터를 갖기 때문에 CPU들에 분산되어 있는 쓰레드들은 지역 카운터를 경쟁 없이 갱신할 수 있다. 그러므로 카운터 갱신은 확장성이 있다.

쓰레드가 카운터의 값을 읽을 수 있기 때문에 전역 카운터를 최신으로 갱신해야 한다. 최신 값으로 갱신하기 위해서 지역 카운터의 값은 주기적으로 전역 카운터로 전달되는데, 이때 전역 락을 사용하여 지역 카운터의 값을 전역 카운터의 값에 더하고, 그 지역 카운터의 값은 0으로 초기화한다.

지역에서 전역으로 값을 전달하는 빈도는 정해 놓은 S(sloppiness의 앞 글자) 값에 의해 결정된다. S의 값이 작을수록 확장성 없는 카운터처럼 동작하며, 커질수록 전역 카운터의 값은 실제 값과 차이가 있게 된다. 정확한 카운터 값을 얻기 위해 모든 지역 락과 전역 락을 획득한 후 계산을 하면 되지만, 확정성이 없게 된다. 이 경우에, 지역 락을 획득하는 순서를 적절히 제어하지 않으면 교착 상태가 발생한다.

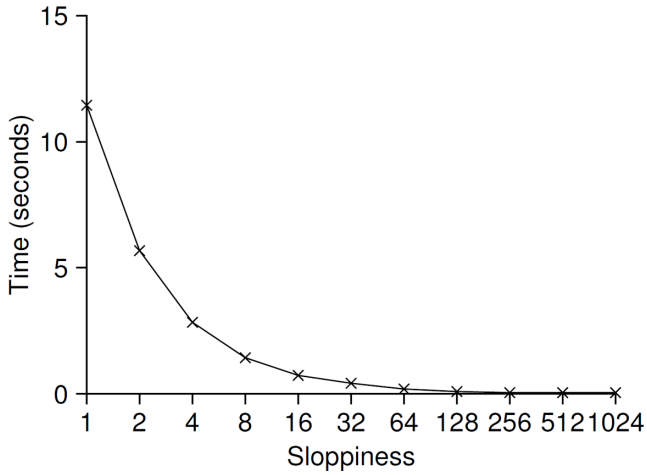
그림 32.4에 예제를 살펴보자. 이 예제에선 한계치를 5로 설정하였고 네 개의 CPU에 각각의 지역 카운터 L_1 ... L_4를 갱신하는 쓰레드들이 있다. 전역 카운터의 값(G)도 같이 나타내었는데, 만약 지역 카운터가 한계치 S에 도달하면 지역 값은 전역 카운터에 반영되고 지역 카운터의 값은 초기화된다.

시간	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 (L_1 에서)
7	0	2	4	5→0	10 (L_4 에서)

〈그림 32.4〉 **엉성한 카운터의 흐름**

그림 32.3에서 범례에 sloppy라고 적힌 아래 선이 S 값을 1024로 했을 때 엉성한 카운터의 성능이다. 탁월한 성능을 보이고 있다. 네 개의 프로세서에서 카운터의 값을 4백만 번 갱신하는 데 걸린 시간이 하나의 프로세서에서 백만 번 갱신하는 시간에 비해 그리 크지 않다.

그림 32.5는 4개의 CPU에서 4개의 쓰레드가 각각 백만 번 카운터의 값을 증가하는 경우에 한계치 S의 설정이 얼마나 중요한지를 보여준다. S의 값이 낮다면 성능이 낮은 대신 전역 카운터의 값이 매우 정확해진다. S의 값이 매우 크다면 성능은 탁월하겠지만 전역 카운터의 값은 CPU의 개수와 S의 곱만큼 뒤처지게 된다. 엉성한 카운터는 정확도와 성능의 균형을 조절할 수 있다.



〈그림 32.5〉 **엉성한 카운터의 확장성**

엉성한 카운터의 대략적인 코드를 그림 32.6에 나타내었다. 읽어보기 바란다. 더 좋은 방법은 이 기법을 사용하는 실험을 해보고 어떻게 동작하는지 자세히 이해하는 것이다.

32.2 병행 연결 리스트

이제 조금 더 복잡한 구조인 연결 리스트를 다뤄보자. 기본적인 것을 갖고 시작해 보자. 간단하게 하기 위해서, 병행 삽입 연산만 살펴보자. 검색, 삭제와 같은 연산은 독자의 몫으로 남겨 두겠다. 그림 32.7은 자료 구조의 기본이 되는 코드이다.

코드에서 볼 수 있듯이, 삽입 연산을 시작하기 전에 락을 획득하고 리턴 직전에 해제한다. 매우 드문 경우지만, `malloc()`이 실패할 경우에 미묘한 문제가 생길 수 있다. 그런 경우가 발생한다면 실패를 처리하기 전에 락을 해제해야 한다.

예외 처리에서는 쉽게 오류가 발생하는 것으로 알려져 있다. 최근 Linux 커널 패치에 관한 연구에 따르면 약 40%에 달하는 버그가 자주 사용되지 않는 이런 코드에서 발생된 것으로 조사되었다 (이 결과에서 영감을 얻어, 우리는 메모리 할당 실패를 발생시키는 부분을 제거한 안정적인 파일 시스템을 만들었다 [Sun+11]).

다음을 해결해 보자. 삽입 연산이 병행하여 진행되는 상황에서 실패를 하더라도 락 해제를 호출하지 않으면서 삽입과 검색이 올바르게 동작하도록 수정할 수 있을까?

```
1  typedef struct __counter_t {
2      int global; // 전역 카운터
3      pthread_mutex_t glock; // 전역 카운터
4      int local[NUMCPUS]; // 지역 카운터 (cpu당)
5      pthread_mutex_t llock[NUMCPUS]; // ... 그리고 락들
6      int threshold; // 갱신 빈도
7  } counter_t;
8
9  // init:한계치를 기록하고 락과 지역 카운터
10 // 그리고 전역 카운터의 값들을 초기화함
11 void init(counter_t *c, int threshold) {
12     c->threshold = 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
24 // update: 보통은 지역 락을 획득한 후 지역 값을 갱신함
25 //          '한계치' 까지 지역 카운터의 값이 커지면,
26 //          전역 락을 획득하고 지역 값을 전역 카운터에 전달함
27 void update(counter_t *c, int threadID, int amt) {
28     pthread_mutex_lock(&c->llock[threadID]);
29     c->local[threadID] += amt; // amt > 0을 가정
30     if (c->local[threadID] >= c->threshold) { // 전역으로 전달
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: 전역 카운터의 값을 리턴 (정확하지 않을 수 있음)
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; // 대략의 값임!
45 }
```

〈그림 32.6〉 **엉성한 카운터의 구현**

답은 가능하다이다. 삽입 코드에서 임계 영역을 처리하는 부분만 락으로 감싸도록 코드 순서를 변경하고, 검색 코드의 종료는 검색과 삽입 모두 동일한 코드 패스를 사용토록 할 수 있다. 이 방법이 동작하는 이유는 검색 코드의 일부는 사실 락이 필요 없기 때문이다. `malloc()` 자체가 쓰레드 안전하다면 쓰레드는 언제든지 경쟁 조건과 다른 병행성 관련 버그를 걱정하지 않으면서 검색할 수 있다. 공유 리스트 갱신 때에만 락을 획득하면 된다. 변경된 세부 내용을 이해하기 위해 그림 32.8을 보자.

검색 루틴의 while 문 안에 break를 삽입하여 검색이 성공하면 바로 빠져나오게 수정한다. 이렇게 하면 검색 성공이나 실패의 경우 모두 동일한 리턴 코드를 실행한다. 이렇게 하면 코드에서 락을 획득하고 해제하는 문장 수를 줄일 수 있다. 버그(락을 해제하지 않고 리턴하는 것과 같은) 발생 여지가 줄어 든다.

```
1  // 기본 노드 구조
2  typedef struct __node_t {
3      int key;
4      struct __node_t *next;
5  } node_t;
6
7  // 기본 리스트 구조 (리스트마다 하나씩 사용)
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 }
17
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");
23         pthread_mutex_unlock(&L->lock);
24         return -1;   // 실패
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0;         // 성공
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; // 성공
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1;         // 오류
45 }
```

〈그림 32.7〉 **병행 연결 리스트**

확장성 있는 연결 리스트

병행이 가능한 연결 리스트를 갖게 되었지만 확장성이 좋지 않다는 문제가 있다. 병행성을 개선하는 방법 중 하나로 **hand-over-hand locking**(또는 **lock coupling**) [MS04]이라는 기법을 개발했다.

개념은 단순하다. 전체 리스트에 하나의 락이 있는 것이 아니라 개별 노드마다 락을 추가하는 것이다. 리스트를 순회할 때 다음 노드의 락을 먼저 획득하고 지금 노드의 락을 해제하도록 한다(hand-over-hand라는 이름에 영감이 되었다).

개념적으로는, 리스트 연산에 병행성이 높아지기 때문에 괜찮은 것처럼 보인다. 하지만 실제로는 간단한 락 방법에(락을 하나 두는) 비해 속도 개선을 기대하기가 쉽지 않다. 리스트를 순회할 때 각 노드에 락을 획득하고 해제하는 오버헤드가 매우 크기

```
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      // 동기화할 필요 없음
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // 임계 영역만 락으로 보호
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
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; // 성공과 실패를 나타냄
35 }
```

〈그림 32.8〉 **병행 연결 리스트 : 개선함**

팁 : 병행성이 늘어난다고 더 빠른 것은 아니다

락 획득/해제와 같이 부하가 큰 연산을 추가하여 자료 구조를 설계했다면, 병행성 자체가 좋아졌다는 것은 큰 의미가 없다. 오히려, 부하가 큰 루틴은 거의 사용하지 않는 간단한 방법이 더 좋다. 락을 많이 추가하고 복잡도가 증가하면 큰 단점이 된다. 어느 것이 더 좋은지 알 수 있는 방법은 딱 한 가지이다. 간단하지만 병행성이 떨어지는 것과 복잡하지만 병행성이 높은 두 경우를 다 구현하고 성능을 측정해 보아야 한다. 성능을 속일 수는 없지 않는가. 결과는 "좋다", "나쁘다" 둘 중에 하나일 테니까 말이다.

때문이다. 아주 큰 리스트를 굉장히 많은 수의 쓰레드가 병행적으로 순회한다해도, 락을 하나만 사용하는 것보다 빠르기 어렵다. 차라리 일정 개수의 노드를 처리할 때마다 하나의 새로운 락을 획득하는 하이브리드 방식이 더 가치 있어 보인다.

팁: 락과 제어 흐름을 경계하자

병행 코드를 포함하여 일반적인 코드를 작성할 때, 함수의 흐름이 바뀌는 부분, 즉 return, exit, 에러로 인한 실행 중지에 매우 세심한 주의를 기울여야 한다. 왜냐하면, 많은 함수들이 락 획득, 메모리 할당, 또는 상태를 변경하는 연산들을 실행하는 데 에러가 발생하면 리턴하기 전에 이들을 이전 상태로 복구해야 하기 때문이다. 이 과정에서 에러가 발생하기 쉽다. 그렇기 때문에 이런 패턴이 발생하지 않도록 구조적으로 코드를 작성하는 것이 좋다.

32.3 병행 큐

이제 알게 되었겠지만, 큰 락을 사용하는 것이 병행 자료 구조를 만들기에 표준이다. 큐에서는 이 방법을 사용하지 않을 것이다. 이유는 쉽게 알 수 있을 테니, 긴 말 않겠다.

대신에 Michael과 Scott [MS98]이 설계한 조금 더 병행성이 좋은 큐에 대해서 살펴보도록 하겠다. 이 큐가 사용하는 자료 구조와 코드는 그림 32.9에 나타나 있다.

이 코드를 유심히 살펴보면 두 개의 락이 있는데, 하나는 큐의 헤드에, 다른 하나는 테일에 사용되는 것을 알 수 있다. 이 두 개 락의 목적은 큐에 삽입과 추출 연산에 병행성을 부여하는 것이다. 일반적인 경우에는 삽입 루틴이 테일 락을 접근하고 추출 연산이 헤드 락만을 다룬다.

Michael과 Scott은 큐 초기화 코드에 더미(dummy) 노드 하나를 추가했다. 이 더미 노드는 헤드와 테일 연산을 구분하는 데 사용되었다. 코드를 이해해 보라. 더 좋은 방법은 코드로 작성해서 실행하고, 성능을 측정하여 어떻게 동작하는지를 심도있게 분석해 보는 것이다.

큐는 멀티 쓰레드 프로그램에서 자주 사용된다. 하지만 방금 다룬 락만 있는 큐는 그런 프로그램에서 사용할 수가 없다. 큐가 비었거나 가득 찬 경우, 쓰레드가 대기하도록 하는 기능이 필요하다. 유한 큐를 만드는 법에 대해 다음 장에서 다룰 것이다. 주제는 바로 그 유명한 조건 변수(condition variable)이다. 기대하시라.

32.4 병행 해시 테이블

많이 사용되는 병행 자료 구조인 해시 테이블을 다루면서 이번 장의 논의를 마무리 짓고자 한다(그림 32.10). 확장이 되지 않는 간단한 해시 테이블을 집중하겠다. 동적으로 크기가 변하는 것은 그렇게 어렵지 않기 때문에 독자의 몫으로 남긴다(미안!).

이 병행 해시 테이블은 명확하다. 이전에 학습한 병행 리스트를 사용하여 구현하였으며 잘 동작한다. 성능이 우수한 이유는 전체 자료 구조에 하나의 락을 사용한 것이 아니라 해시 버켓(리스트로 구현되어 있음)마다 락을 사용하였기 때문이다. 이렇게 하면 병행성이 좋아진다.

```
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
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; // 큐가 비어 있음
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

〈그림 32.9〉 **Michael과 Scott의 병행 큐**

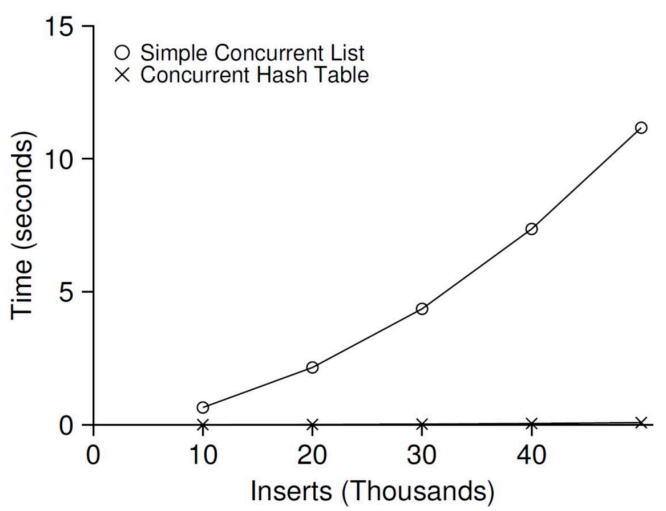
그림 32.11은 4개의 CPU를 갖는 iMac에서 4개의 쓰레드가 해시 테이블에 각각 10,000개에서 50,000개의 갱신 연산을 수행할 때의 성능을 나타낸다. 비교를 위해서 단일 락을 사용하는 연결 리스트의 성능도 함께 보였다. 그림에서 볼 수 있듯이 병행 해시 테이블은 확장성이 매우 좋다. 반면에 연결 리스트는 확장성이 매우 떨어진다.

32.5 요약

카운터, 리스트, 큐 그리고 자주 쓰이는 해시 테이블이라는 병행 자료 구조들 몇 가지를 소개하였다. 소개 중에 몇 가지 중요한 교훈을 얻었다. 락 획득과 해제 시 코드의 흐름에 매우 주의를 기울여야 한다는 것과 병행성 개선이 반드시 성능 개선으로 이어지는 것은

```
1  #define BUCKETS ()
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 = ; 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 }
```

〈그림 32.10〉 **병행 해시 테이블**



〈그림 32.11〉 **확장성이 좋은 해시 테이블**

아니라는 것, 그리고 성능 개선은 성능에 문제가 생길 경우에만 해결책을 간구해야 한다는 것이다. 마지막에 언급한 **미숙한 최적화(premature optimization)**를 피하자는 주제는 여느 성능에 관심있는 개발자라면 중심이 되는 주제이다. 부분적인 성능 개선 시도가 응용 프로그램의 전체 성능을 개선하지 못하면 아무 의미가 없다.

고성능 자료 구조들을 피상적으로만 다루었다[1]. 더 자세한 사항은 Moir와 Shavit의 훌륭한 개관적 연구와 다른 자료에 대한 소개를 살펴보자 [MS04]. 특히, B-tree와 같은 다른 자료 구조들에 흥미를 느낄 수 있을 것이다. 이를 위해서는 데이터베이스

1) 당연하다. 이 책은 학부생용 교재다.

팁: 미숙한 최적화를 피하자(Knuth's Law)

병행 자료 구조를 만들 때는 하나의 큰 락을 추가하여 동기화 접근을 제어하는 가장 간단한 방법에서 시작하자. 그렇게 하면 제대로 동작하는 락을 만들 수 있게 된다. 성능 하락이 문제된다고 판단이 되면 그때 개선을 하자. 결과적으로 꼭 필요한 만큼만 빠르게 만들 수 있다. **Knuth**가 했던 유명한 말처럼, "미숙한 최적화는 모든 악의 근원이다."

Sun 운영체제와 Linux를 포함한 많은 운영체제가 처음 멀티프로세서 용으로 변화할 때 하나의 락을 사용했었다. 이런 종류의 락은 **큰 커널 락**(**big kernel lock**, **BKL**)이라는 이름을 갖게 되었다. 오랜 기간 동안, 이 기법만으로도 충분했다. 멀티 CPU 시스템이 대중화되면서 하나의 쓰레드만 커널에 있을 수 있는 것이 성능에 큰 병목이 되었다. BKL은 2011년에 마침내 Linux 운영체제에서 재거되었다. 이 시스템들에 병행성을 개선해야 할 시기가 되었다. Linux에서는 하나의 락을 여러 개의 락으로 대체하는 가장 자명한 방법을 채용하였다. BSD의 아류인 SunOS에서, 기존의 락을 작은 단위의 락으로 대체하는 것이 보통 어려운 작업이 아니었다. Sun에서는 좀 더 과감한 전략을 택하였다. 초기 설계부터 병행성을 고려하여 운영체제를 개발하였다. 이렇게해서 탄생한 것이 Solaris 운영체제이다. Linux와 Solaris에 대한 책을 읽어 보아라 [BC05; MM00]

수업을 들어야 한다. 락을 전혀 사용하지 않는 동기화 기법들도 매우 재미있다. 그런 **non-blocking 자료 구조**들은 병행성 관련 버그를 다루는 장에서 살펴볼 것이다. 하지만, 솔직히 non-blocking 병행 제어에 관련된 주제는 이 책에 있는 내용을 훨씬 뛰어넘는, 다양한 전문 지식을 필요로 한다. 흥미를 느낀다면 스스로 좀 더 찾아보자(항상 그랬듯이!).

참고 문헌

[BC05] **"Understanding the Linux Kernel (Third Edition)"**
Daniel P. Bovet and Marco Cesati
O'Reilly Media, November 2005
Linux 커널에 대한 고전이다. 이 책은 꼭 읽어야 한다.

[Boy+10] **"An Analysis of Linux Scalability to Many Cores"**
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
OSDI '10, Vancouver, Canada, October 2010
멀티코어 기기에서 Linux의 성능과 간단한 해법들을 다룬 훌륭한 연구이다.

[Han73] **"Operating System Principles"**
Per Brinch Hansen
Prentice-Hall, 1973
URL: `http://portal.acm.org/citation.cfm?id=540365`
시대에 앞선 운영체제에 대한 초기 서적 중 하나이다. 병행성 기법의 하나로 모니터 기법을 소개하였다.

[MM00] **"Solaris Internals: Core Kernel Architecture"**
Jim Mauro and Richard McDougall
Prentice Hall, October 2000
Solaris 책이다. Linux 외의 다른 것에 대해 상세히 배우고 싶다면 이 책도 반드시 읽어야 하는 책이다.

[MS98] **"Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared memory Multiprocessors"**
M. Michael and M. Scott
Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998
스캇(Scott) 교수와 그의 학생들은 몇 년 동안 병행 알고리즘과 자료 구조 분야에 선두였다. 더 자세한 내용을 위해 그의 홈페이지와 여러 연구 논문들 그리고 책들을 찾아보자.

[MS04] **"Concurrent Data Structures"**
Mark Moir and Nir Shavit
In Handbook of Data Structures and Applications (Editors D. Metha and S. Sahni) Chapman and Hall/CRC Press, 2004
URL: `www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf`
병행 자료 구조에 대한 짧지만 비교적 포괄적인 참고 문헌이다. 이 분야의 최신 기법들은 없지만 (오래 전에 출판되었다), 아직도 믿기지 않을 정도로 유용한 참고 문헌으로 남아 있다.

[Sun+11] **"Making the Common Case the Only Case with Anticipatory Memory Allocation"**
Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '11, San Jose, CA, February 2011
커널의 코드 흐름 중 `malloc()`을 호출할 때 오류가 발생할 수 있는 부분들을 제거한 우리의 연구 논문이다. 어떤 작업을 수행하기 전에 잠재적으로 필요한 모든 메모리를 먼저 할당받아 스토리지 스택 깊은 곳에서 오류가 발생하는 것을 막자는 것이 주요 아이디어이다.