33

컨디션 변수

지금까지 락의 개념을 학습하면서 하드웨어와 운영체제의 적절한 지원을 통해 제대로 된 락을 만드는 법을 살펴보았다. 불행히도 "락"만으로는 병행 프로그램을 제대로 작성할 수 없다.

쓰레드가 계속 진행하기 전에 어떤 **조건**이 참인지를 검사해야 하는 경우가 많이 있다. 예를 들어 부모 쓰레드가 작업을 시작하기 전에 자식 쓰레드가 작업을 끝냈는지를 검사하기를 원할 수 있다(이 과정을 보통 `join()`이라 부른다). 그런 대기문은 어떻게 구현해야 할까? 그림 33.1을 살펴보자.

```
1  void *child(void *arg) {
2      printf("child\n");
3      // 작업이 끝났음을 어떻게 알리지?
4      return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8      printf("parent: begin\n");
9      pthread_t c;
10     Pthread_create(&c, NULL, child, NULL); // 자식을 생성하기
11     // 자식 쓰레드를 어떻게 기다리지?
12     printf("parent: end\n");
13     return 0;
14 }
```

〈그림 33.1〉 **자식을 기다리는 부모 쓰레드**

우리가 다음과 같은 출력문을 원한다.

```
parent: begin
child
parent: end
```

이를 위해 그림 33.2에서처럼 공유 변수를 사용할 수도 있다. 이 방법은 제대로 동작하지만 부모 쓰레드가 회전을 하면서 CPU 시간을 낭비하기 때문에 비효율적이다. 이 방법 대신에 부모 쓰레드가 특정 조건이 참이 될 때까지(예: 자식 쓰레드가 실행이 종료되는 것) 잠자면서 기다리는 방법이 더 좋다.

```
1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // 자식 생성하기
13     while (done == 0)
14     ; // 회전
15     printf("parent: end\n");
16     return 0;
17 }
```

〈그림 33.2〉 **자식을 기다리는 부모 쓰레드: 회전 기반의 방법**

핵심 질문: 조건을 기다리는 법

멀티 쓰레드 프로그램에서는 쓰레드가 계속 진행하기에 앞서 특정 조건이 참이 되기를 기다리는 것이 유용할 때가 많이 있다. 조건이 참이 될 때까지 회전을 하며 기다리는 것이 간단하기는 하겠지만 지독하게 비효율적일 뿐만 아니라 CPU 사이클을 낭비한다. 어떤 경우에는 부정확할 수도 있다. 그렇다면, 쓰레드는 어떻게 조건을 기다려야 할까?

33.1 정의와 루틴들

조건이 참이 될 때까지 기다리기 위해 **컨디션 변수**(**conditional variable**)를 활용할 수 있다. 컨디션 변수는 일종의 큐 자료 구조로서, 어떤 실행의 상태(또는 어떤 **조건**)가 원하는 것과 다를 때 조건이 참이 되기를 기다리며 쓰레드가 **대기**할 수 있는 큐이다. 다른 쓰레드가 상태를 변경시켰을 때, 대기 중이던 쓰레드(하나 이상의 쓰레드가 깨어날 수도 있다)를 깨우고(조건에 따라 **시그널**을 보내어), 계속 진행할 수 있도록 한다. 이 개념은 Dijkstra가 "Private Semaphore"라는 개념을 사용했을 때부터 쓰였다 [Dij68]. 이후에 Hoare가 모니터 [Hoa74]에 대한 연구를 진행하면서 이와 비슷한 개념에 "Condition Variable"이라는 이름을 붙였다.

`pthread_cond_t c;` 라고 써서 c가 컨디션 변수가 되도록 선언한다(적절한 초기화 과정이 필요하다는 것을 주의하자). 컨디션 변수에는 `wait()`와 `signal()`이라는 두 개의 연산이 있다. `wait()`는 쓰레드가 스스로를 잠재우기 위해서 호출하는 것이고 `signal()`은 쓰레드가 무엇인가를 변경했기 때문에 조건이 참이 되기를 기다리며 잠자고 있던 쓰레드를 깨울 때 호출한다. POSIX의 사용례는 다음과 같다.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

각각을 간단히 `wait()`와 `signal()`이라 칭하겠다. `wait()`에서 유의할 것은 `mutex`를 매개변수로 사용한다는 것이다. "`wait()`가 호출될 때 `mutex`는 잠겨있었다

고 가정한다. `wait()`의 역할은 락을 해제하고 호출한 쓰레드를 재우는 것이다. 어떤 다른 쓰레드가 시그널을 보내어 쓰레드가 깨어나면, `wait()`에서 리턴하기 전에 락을 재획득해야 한다." 좀 복잡하다. 그러나 너무 중요하다. 10번만 소리내어 읽고 넘어가기 바란다. `wait()`에서 리턴할 때 락을 재 획득해야 한다! 그러니 조건이 만족되어 잠에서 깨어났더라도 락을 획득못하면 다시 sleep 상태로 들어간다는 말이다. 이렇게 복잡한 이유는 쓰레드가 스스로를 재우려고 할 때, 경쟁 조건의 발생을 방지하기 위해서이다. 이해를 돕기 위해 그림 33.3에 나타난 join 문제의 해법을 살펴보자.

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

〈그림 33.3〉 **자식을 기다리는 부모 쓰레드 : 컨디션 변수의 사용**

두 가지 경우가 있다. 첫 번째는, 부모 쓰레드가 자식 쓰레드를 생성하고, 계속 실행하여 `thr_join()`을 호출하고 자식 쓰레드가 끝나기를 기다리는 경우이다. 이 경우에 부모 쓰레드가 락을 획득하고 자식이 끝났는지 검사한 후에(즉, `done`이 1인지), 자식이 끝나지 않았으므로, `wait()`를 호출하여 스스로를 잠재운다(그리고 락을 해제한다). 자식 쓰레드가 추후에 실행되어 "child"라는 메시지를 출력하고 `thr_exit()`을 호출하여 부모 쓰레드를 깨울 것이다. 이 코드는 락을 획득한 후에 상태 변수를 `done`으로 설정하고 부모 쓰레드에 시그널을 보내어 깨어나도록 한다. 마지막으로, 호출했던 `wait()`에서 락을 획득한 채로 리턴하여 부모 쓰레드가 실행될 것이며 락을 해제하고 "parent: end" 메시지를 출력할 것이다.

두 번째 경우에서는 자식 쓰레드가 생성되면서 즉시 실행되고 `done` 변수를 1로

설정하고, 자고 있는 쓰레드를 깨우기 위해 시그널을 보낸다. 하지만 자고있는 쓰레드가 없기 때문에 단순히 리턴한다. 한 마디로 헛 수고한 셈이다. 그 후에 부모 쓰레드가 실행하고 `thr_join()`을 호출하고 `done` 변수가 1인 것을 알게 된다. `done`이 1이므로 대기 없이 바로 리턴한다.

그리고 매우 중요한 사실이 있다. 부모 쓰레드가 조건을 검사할 때(이 예제에서는 `done`의 값) `if` 문이 아니라 `while` 문을 사용한다는 사실이다! 꼭 이래야 하나 싶겠지만, 이렇게 하는 것이 좋다는 것을 이제 곧 알게 될 것이다. 잠시만 기다리기 바란다.

`thr_exit()`와 `thr_join()` 코드 내용의 중요성을 이해할 수 있도록 몇 가지 구현의 방식을 살펴보자. 먼저, done 이라는 상태 변수가 꼭 필요할까? 아래 예제와 같이 코드가 작성되었다면 어떨까? 이 코드가 동작하겠는가?

```
1  void thr_exit() {
2      Pthread_mutex_lock(&m);
3      Pthread_cond_signal(&c);
4      Pthread_mutex_unlock(&m);
5  }
6
7  void thr_join() {
8      Pthread_mutex_lock(&m);
9      Pthread_cond_wait(&c, &m);
10     Pthread_mutex_unlock(&m);
11 }
```

불행하게도 이 방법은 틀렸다. 자식 쓰레드가 생성된 즉시 실행되어서 `thr_exit()`를 호출하는 경우를 생각해 보자. 그 경우에, 자식 프로세스가 시그널을 보내겠지만, 깨워야 할 쓰레드가 없다. 부모 쓰레드가 실행되면 `wait()`를 호출하고 거기서 멈춰 있을 것이다. 어떤 쓰레드도 부모 쓰레드를 깨우지 않을 것이다. 이 예제를 통해, 다른 쓰레드들이 알고자 하는 정보를 기록하는 done이라는 상태 변수의 필요성을 알 수 있다. 잠자고, 깨우고, 락을 설정하는 것이 done이라는 상태 변수를 중심으로 구현되어 있다.

또 다른 안 좋은 예가 있다. 시그널을 주거나 대기할 때 락을 획득할 필요가 없다고 가정해 보자. 어떤 문제가 생길까? 생각해 보라!

```
1  void thr_exit() {
2      done = 1;
3      Pthread_cond_signal(&c);
4  }
5
6  void thr_join() {
7      if (done == 0)
8          Pthread_cond_wait(&c);
9  }
```

여기서는 경쟁 조건이 발생한다. 만약 부모 쓰레드가 `thr_join()`을 호출하고 나서 done 변수의 값이 0인 것을 확인한 후 잠자려고 한다. 하지만 `wait()`를 호출하기 직전에 부모 쓰레드가 인터럽트에 걸려서 자식 쓰레드가 실행이 되었다. 자식 쓰레드는 상태 변수 done의 값을 1로 변경하고 시그널을 보낸다. 하지만 대기 중인 쓰레드가 없다. 부모 쓰레드가 다시 실행되면, `wait()`를 호출하고 잠자게 된다. 문제는 잠든 부모 쓰레드를 깨워줄 쓰레드가 없다는 것이다.

이 간단한 join 예제를 통해서, 컨디션 변수를 제대로 사용하기 위한 기본적인 요건들

팁: 시그널을 보내기 전에 항상 락을 획득하자

모든 경우에 꼭 락을 획득할 필요는 없지만, 컨디션 변수를 사용할 때는 락을 획득한 후에 시그널을 보내는 것이 가장 간단하고 최선의 방법이다. 올바른 프로그램 작동을 위해 락을 반드시 획득해야 하는 경우를 예제에서 보았다. 락을 획득하지 않아도 괜찮은 경우들이 있기는 하다. 하지만, 속 편하게 **시그널을 보낼 때는 락을 무조건 획득하자.**

`wait()`를 호출할 때 락을 획득하는 것은 단순한 권고 사항이 아니며 `wait()`의 정의상 무조건 해야 한다. 왜냐하면 `wait()`는 항상 (a) `wait()`를 호출했을 때 락을 갖고 있다고 가정하며, (b) 호출자를 잠들게 할 때 락을 해제하고, (c) 리턴하기 직전에 락을 다시 획득한다. 정리하면, **시그널을 보낼 때, 대기할 때 항상 락을 걸자!**

을 이해했기 바란다. 확실히 이해할 수 있도록, 좀 더 복잡한 예제를 살펴보도록 하겠다. **생산자/소비자** 또는 **유한 버퍼** 문제이다.

33.2 생산자/소비자(유한 버퍼) 문제

다음으로 살펴볼 동기화 문제는 Dijkstra가 처음 제시한 **생산자/소비자(producer/consumer)** 문제이다. **유한 버퍼(bounded 버퍼)** 문제로도 알려져 있다 [Dij72]. Dijkstra와 그의 동료들이 락이나 컨디션 변수를 대신하여 사용할 수 있는 일반화된 세마포어를 발명하게 된 이유가 이 생산자/소비자 문제 때문이다 [Dij01]. 세마포어에 대해서는 차차 다루게 될 것이다.

여러 개의 생산자 쓰레드와 소비자 쓰레드가 있다고 하자. 생산자는 데이터를 만들어 버퍼에 넣고, 소비자는 버퍼에서 데이터를 꺼내어 사용한다.

이러한 관계는 실제 시스템에서 자주 일어난다. 예를 들어 멀티 쓰레드 웹 서버의 경우 생산자는 HTTP 요청을 작업 큐(즉, 유한 버퍼)에 넣고, 소비자 쓰레드는 이 큐에서 요청을 꺼내어 처리한다.

`grep foo file.txt | wc -l`과 같은 문장처럼 파이프(pipe) 명령으로 한 프로그램의 결과를 다른 프로그램에게 전달할 때도 유한 버퍼를 사용한다. 이 예제는 두 개의 프로세스가 병행 실행된다. `grep` 명령어는 `file.txt`에서 `foo`라는 문자열이 포함된 줄만을 찾아 표준 출력(standard ouput, STDOUT)에 쓴다. Unix 쉘은 출력 결과를 Unix 파이프(시스템 콜인 파이프(pipe)에 의해 생성)라는 곳으로 전송(redirect)한다. 파이프의 한쪽 끝에는 `wc` 프로세스의 표준 입력과 연결되어 있다. `grep` 프로세스가 생산자가 되고 `wc` 프로세스가 소비자가 된다. 그 둘 사이에는 커널 내부에 있는 유한 버퍼가 있다. 이 예제에서 당신은 별 생각 없는 사용자일 뿐이다.

유한 버퍼는 공유 자원이다. 경쟁 조건의 발생을 방지하기 위해 동기화가 필요하다. 이 문제를 좀 더 잘 이해하기 위해서 실제 코드를 살펴보자.

먼저, 생산자는 넣고 소비자는 꺼내어 쓸 수 있는 공유 버퍼가 하나 필요하다. 한 개의 정수를 사용하고(물론, 이곳에 다른 구조체를 가리키는 포인터를 넣을 수도 있다),

공유 버퍼에 값을 넣는 루틴과 버퍼에서 값을 꺼내는 루틴 두 개가 있다. 자세한 내용은 그림 33.4에 나와 있다.

```
1  int buffer;
2  int count = 0; // 처음에는 비어있음
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

〈그림 33.4〉 put()과 get() 루틴 (버전 1)

간단하지 않은가? put() 루틴은 버퍼가 비어 있다고 가정하고(assert()문으로 확인도 함), 값을 공유 버퍼 넣은 후에 count를 1로 설정하여 가득 찼다고 표시한다. get() 루틴은 그 반대로 동작하는데, 버퍼가 찼는지 확인하고, 값을 꺼낸 후 버퍼가 비었다고 설정한다(즉, count를 0으로 설정). 읽은 값은 리턴한다. 공유 버퍼에 숫자 한 개만 저장할 수 있다고 걱정하지 말자. 나중에 여러 값을 저장할 수 있는 큐로 일반화할 예정이다. 상당히 재미있는 작업이 될 것이다.

이제 버퍼에 데이터를 넣거나 버퍼의 데이터를 꺼내도 괜찮은지를 판단하는 루틴을 작성해야 한다. 판단하는 조건은 명확하다. 버퍼의 count가 0이면(즉, 버퍼가 비어 있다면) 데이터를 넣고, count가 1일 때만(즉, 버퍼가 가득 찼을 때만) 버퍼에서 데이터를 꺼낸다. 만약 생산자가 가득 찬 버퍼에 데이터를 넣고, 소비자가 비어 있는 버퍼에서 데이터를 꺼내는 동기화 코드를 작성했다면 무언가 잘못 작성한 것이다(이 코드에서는 단언문이 실행될 것이다).

이 작업은 두 종류의 쓰레드에 의해 수행될 것이다. 하나는 생산자 쓰레드들이고 다른 하나는 소비자 쓰레드들이다. 그림 33.5는 생산자가 loop 횟수 만큼 공유 버퍼에 정수를 넣고, 소비자는 (무한히) 데이터를 공유 버퍼에서 꺼내는 코드를 나타낸다. 소비자 쓰레드는 데이터를 공유 버퍼에서 꺼낼 때 그 값을 매번 출력한다. 이 코드가 제대로 동작하지 않는다는 것쯤은 다 알 것이다.

불완전한 해답

생산자와 소비자가 각 하나씩 있다고 가정한다. 당연히 put()과 get()루틴에는 임계영역이 있으며, put()은 버퍼를 갱신하고 get()은 버퍼에서 읽는다. 코드에 락의 추가만으로 제대로 동작하는 것은 아니다. 무언가가 더 필요하다. 그 무언가는 컨디션 변수라는 것이다. 그림 33.6에서 cond 컨디션 변수 하나와 그것과 연결된 mutex 락을 사용한다.

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     int i;
11     while (1) {
12         int tmp = get();
13         printf("%d\n", tmp);
14     }
15 }
```

〈그림 33.5〉 **생산자/소비자 쓰레드 (버전 1)**

```
1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);             // p1
8          if (count == 1)                         // p2
9              Pthread_cond_wait(&cond, &mutex);   // p3
10         put(i);                                 // p4
11         Pthread_cond_signal(&cond);             // p5
12         Pthread_mutex_unlock(&mutex);           // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);             // c1
20         if (count == 0)                         // c2
21             Pthread_cond_wait(&cond, &mutex);   // c3
22         int tmp = get();                        // c4
23         Pthread_cond_signal(&cond);             // c5
24         Pthread_mutex_unlock(&mutex);           // c6
25         printf("%d\n", tmp);
26     }
27 }
```

〈그림 33.6〉 **생산자/소비자: 단일 컨디션 변수와 `if` 문**

생산자와 소비자 사이에 시그널을 보내는 과정을 살펴보자. 생산자는 버퍼가 빌 때까지 기다린다($p1$-$p3$). 소비자도 버퍼가 차기를 기다린다($c1$-$c3$).

생산자와 소비자가 각 하나씩인 경우에 그림 33.6의 코드는 동작한다. 하지만, 두 개 이상의 같은 종류의 쓰레드가 있다면(예를 들어 두 개의 소비자), 이런 해법에는 두 가지 문제가 있다. 무엇이겠는가?

... (잠시 생각해 보자) ...

첫 번째 문제점을 살펴보자. 대기 명령 전의 `if` 문과 관련이 있다. T_{c1}과 T_{c2}라는 두 개의 소비자가 있고 T_p라는 생산자가 하나 있다고 가정하자. 소비자 (T_{c1})가 먼저 실행된다. 락($c1$)을 획득하고 버퍼를 소비할 수 있는지 검사한다($c2$). 그리고 비어

있음을 확인한 후에 대기하며($c3$) 락을 해제한다.

그리고 생산자(T_p)가 실행된다. 락을 획득하고($p1$) 버퍼가 비었는지 확인한다($p2$). 비었음을 발견하고, 버퍼를 채운다($p4$). 생산자는 버퍼가 가득 찼다는 시그널을 보낸다($p5$). 대기 중인 첫째 소비자(T_{c1})는 깨어나 준비 큐(ready queue)로 이동한다. T_{c1}은 이제 실행할 수 있는 상태이지만 아직 실행 상태는 아니다. 생산자는 실행을 계속한다. 버퍼가 차 있으므로 대기 상태로 전이한다($p6$, $p1$-$p3$).

여기에서 문제가 발생한다. 다른 소비자(T_{c2})가 끼어들어서 실행하면서 버퍼 값을 소비한다($c1$, $c2$, $c4$, $c5$, $c6$을 수행, $c3$은 버퍼가 가득 찼기 때문에 건너뜀). T_{c1}이 실행된다고 해보자. 대기에서 리턴하기 전에 락을 획득한다. 그리고 `get()`을 호출하지만($c4$) 버퍼는 비었다! 코드는 의도한 대로 기능하지 못했다. 생산자가 버퍼에 넣어 둔 값을 T_{c2}가 끼어들어서 소비하였기 때문에 T_{c1}이 비어 있는 버퍼를 읽는 행위를 막았어야 했다. 그림 33.7에 각 쓰레드의 동작과 스케줄러의 상태(준비, 실행, 또는 대기)를 시간에 따라 나타내었다.

문제의 원인은 단순하다. T_{c1}이 깨어나서 실행되기까지 사이에 유한 버퍼의 상태가

T_{c1}	상태	T_{c2}	상태	T_p	상태	개수	비고
$c1$	실행		준비		준비	0	
$c2$	실행		준비		준비	0	
$c3$	대기		준비		준비	0	꺼낼 것 없음
	대기		준비	$p1$	실행	0	
	대기		준비	$p2$	실행	0	
	대기		준비	$p4$	실행	1	버퍼 가득 참
	준비		준비	$p5$	실행	1	T_{c1}
	준비		준비	$p6$	실행	1	
	준비		준비	$p1$	실행	1	
	준비		준비	$p2$	실행	1	
	준비		준비	$p3$	대기	1	버퍼 가득 참; 대기
	준비	$c1$	실행		대기	1	T_{c2}가 끼어듦
	준비	$c2$	실행		대기	1	
	준비	$c4$	실행		대기	0	... 데이터를 꺼냄
	준비	$c5$	실행		준비	0	T_p 깨어남
	준비	$c6$	실행		준비	0	
$c4$	실행		준비		준비	0	오, 이런! 데이터가 없음

〈그림 33.7〉 **쓰레드 흐름: 불완전한 해법 (버전 1)**

변경되었다. 시그널은 쓰레드를 깨우기만 한다. 상태가 변경되었을 수 있다는 일종의 힌트에 불과하다(이 경우에서 상태는 버퍼에 값이 추가되었다는 것을 알림). 깨어난 쓰레드가 실제 실행되는 시점에도 그 상태가 유지된다는 보장은 없다. 이런 식으로 시그널을 정의하는 것을 **Mesa semantic**이라 한다. Mesa semantic 이라는 용어는 이 방식으로 컨디션 변수를 구현했던 논문을 따라서 명명되었다 [BW 80]. 대비되는 개념은 **Hoare semantic**이 있다. 구현하기는 더 어렵지만 깨어난 즉시 쓰레드가 실행되는 것을 보장한다 [Hoa74]. 대부분의 시스템이 Mesa semantic을 채용하고 있다.

개선된, 하지만 아직도 불완전한: `if` 문 대신 `while` 문

이 문제는 쉽게 해결할 수 있다(그림 33.8). `if` 문을 `while` 문으로 바꾸면 된다. 이 방법이 왜 동작 가능한지 생각해 보라. 소비자 T_{c1}이 깨어나서(락을 획득한 상태), 즉시 공유 변수의 상태를 재확인한다($c2$). 만약 이 시점에 버퍼가 비어 있다면, 소비자는 대기 상태로 돌아간다($c3$). 당연하겠지만 생산자에서도 `if` 문이 `while` 문으로 변경되었다 ($p2$).

Mesa semantic의 컨디션 변수에서 가장 기본적인 법칙은 **언제나 while 문을 사용**하라는 것이다. 때로는 조건을 재확인하지 않아도 되지만 항상 검사하는 것이 안전하다. 무조건 다시 검사하고 마음 편히 살자.

하지만 아직도 코드에는 버그가 있다. 먼저 언급했던 두 가지 문제 중에 두 번째이다. 알아냈는가? 컨디션 변수가 하나뿐이라는 사실과 관계가 있다. 문제가 무엇인지 계속 읽어보기 전에 찾아보라. 찾아보자!

```
1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          while (count == 1)                     // p2
9              Pthread_cond_wait(&cond, &mutex);  // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&cond);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         while (count == 0)                     // c2
21             Pthread_cond_wait(&cond, &mutex);  // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&cond);            // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

〈그림 33.8〉 **생산자/소비자: 단일 컨디션 변수와 `while` 문**

... (다시 한 번 생각해 보자, 생각이 잘 안 떠오르면 눈을 잠시 감아보자) ...

정답이 맞는지 확인해 보자, 아니면 이제 그만 졸고 다시 책을 읽기 시작하자. 이 문제는 소비자 쓰레드 T_{c1}과 T_{c2}가 먼저 실행한 후에 둘 다 대기 상태에 있을 때 발생한다($c3$). 생산자가 실행되어 버퍼에 값을 넣고 대기 중인 쓰레드 하나를 깨우고 (T_{c1}을 깨웠다고 하자), 자신은 대기한다. 이제 하나의 생산자(T_{c1})가 실행할 준비가 되었고 조건에 의해 T_{c2}와 T_p는 대기 중이다. 이제 문제가 발생하도록 만들 것이다. 점점 흥미진진해진다!

소비자 T_{c1}이 `wait()`에서 리턴을 받아 깨어나고($c3$) 조건을 재확인한다($c2$). 버퍼가 차있다는 것을 발견하고 값을 소비한다($c4$). 이 소비자는 시그널을 전송하여($c5$) 대기 중인 쓰레드 중 하나를 깨운다. 이때 어떤 쓰레드를 깨울 것인가?

소비자가 버퍼를 비웠기 때문에 생산자를 당연히 깨워야 한다. 하지만, 만약 소비자 T_{c2}를 깨운다면(대기 큐가 어떻게 관리되느냐에 따라 당연히 발생할 수 있다), 문제가 발생한다. 소비자 T_{c2}가 깨어나면 버퍼가 비어 있다는 것을 발견한 후에($c2$) 다시 대기 상태로 들어간다($c3$). 버퍼에 값을 넣어야 하는 생산자 T_p는 대기 중이다. 다른 소비자 쓰레드 T_{c1} 역시 대기 상태에 들어간다. 세 개의 쓰레드가 모두 대기 상태다. 이 대형 사고의 발생 과정을 그림 33.9에서 한 단계씩 설명하고 있다.

시그널을 보내는 것은 꼭 필요하지만 대상이 명확해야 한다. 소비자는 다른 소비자를 깨울 수 없고 생산자만 깨워야 하며, 반대로 생산자의 경우도 마찬가지다.

단일 버퍼 생산자/소비자 해법

이 문제에 대한 해법 역시 약간의 변경만 필요로 한다. 두 개의 컨디션 변수를 사용하여 시스템의 상태가 변경되었을 때 깨워야 하는 쓰레드에게만 시그널을 제대로 전달한다. 그림 33.10이 변경된 코드를 나타낸다.

앞서 살펴본 코드에서는, 생산자 쓰레드가 `empty` 조건 변수에서 대기하고 `fill`에 대해서 시그널을 발생한다. 정반대로 소비자 쓰레드는 `fill` 에 대해서 대기하고 `empty`에 대해서 시그널을 발생시킨다. 그렇게 함으로써, 두 번째 문제가 발생하는 것을 피했다. 소비자가 실수로 다른 소비자를 절대로 깨울 수 없도록 하였고, 생산자도 다른 생산자를 깨우는 일이 절대 없도록 만들었다.

최종적인 생산자/소비자 해법

이제 제대로 동작하는 생산자/소비자 해법을 얻었지만 아직까지는 보편적인 방법은 아니다. 마지막 변경을 통해 병행성을 증가시키고 더 효율적으로 만든다. 버퍼 공간을 추가하여 대기 상태에 들어가기 전에 여러 값들이 생산될 수 있도록 하는 것, 그리고 마찬가지로 여러 개의 값이 대기 상태 전에 소비될 수 있도록 하는 것이다. 하나의 생산자와 소비자의 경우에서는 버퍼가 커지면 쓰레드 간의 문맥 교환이 줄어들기 때문에 더 효율적이 된다. 멀티 생산자의 경우 또는 멀티 소비자의 경우 (또는 둘 다인 경우)가 되면 생산과 소비가 병행이 될 수 있기 때문에 병행성이 좋아진다. 현재 해법에서 조금만 변경하면 된다.

T_{c1}	상태	T_{c2}	상태	T_p	상태	개수	비고
$c1$	실행		준비		준비	0	
$c2$	실행		준비		준비	0	
$c3$	대기		준비		준비	0	꺼낼 것 없음
	대기	$c1$	실행		준비	0	
	대기	$c2$	실행		준비	0	
	대기	$c3$	대기		준비	0	꺼낼 것 없음
	대기		대기	$p1$	실행	0	
	대기		대기	$p2$	실행	0	
	대기		대기	$p4$	실행	1	버퍼 가득 참
	준비		대기	$p5$	실행	1	T_{c1} 깨어남
	준비		대기	$p6$	실행	1	
	준비		대기	$p1$	실행	1	
	준비		대기	$p2$	실행	1	
	준비		대기	$p3$	대기	1	대기해야 함(가득 참)
$c2$	실행		대기		대기	1	조건 재확인
$c4$	실행		대기		대기	0	T_{c1}이 데이터를 꺼냄
$c5$	실행		준비		대기	0	이런! T_{c2} 가 깨어남
$c6$	실행		준비		대기	0	
$c1$	실행		준비		대기	0	
$c2$	실행		준비		대기	0	
$c3$	대기		준비		대기	0	꺼낼 것 없음
	대기	$c2$	실행		대기	0	
	대기	$c3$	대기		대기	0	모두 대기 중

〈그림 33.9〉 **쓰레드 흐름: 불완전한 해법 (버전 2)**

첫 번째는 그림 33.11에서 다같이 버퍼 구조와 `put()`과 `get()`을 변경하는 것이다. 생산자와 소비자가 대기 상태가 되는지에 대한 여부를 결정하는 조건도 약간 변경하였다. 그림 33.12에서 최종적인 대기와 시그널에 대한 논리를 나타내었다. 생산자는 모든 버퍼가 현재 가득 차있다면 대기 상태에 들어가고($p2$), 마찬가지로, 소비자도 모든 버퍼가 비어 있다면 대기에 들어간다($c2$). 이렇게 생산자/소비자 문제를 해결하였다.

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

〈그림 33.10〉 **생산자/소비자: 두 개의 컨디션 변수와 `while` 문**

```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

〈그림 33.11〉 **`put()`과 `get()` 루틴 최종**

33.3 컨디션 변수 사용 시 주의점

이번에는 컨디션 변수의 사용 예를 하나 살펴보겠다. Lampson과 Redell의 pilot 논문에서 가져온 코드이다. 앞선 설명에서 다뤘던 **Mesa semantic**를 처음 구현했던 연구그룹이다(이름은 Mesa 라는 언어를 사용하였던 것에 기인한다).

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);             // p1
8          while (count == MAX)                    // p2
9              Pthread_cond_wait(&empty, &mutex);  // p3
10         put(i);                                 // p4
11         Pthread_cond_signal(&fill);             // p5
12         Pthread_mutex_unlock(&mutex);           // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);             // c1
20         while (count == 0)                      // c2
21             Pthread_cond_wait(&fill, &mutex);   // c3
22         int tmp = get();                        // c4
23         Pthread_cond_signal(&empty);            // c5
24         Pthread_mutex_unlock(&mutex);           // c6
25         printf("%d\n", tmp);
26     }
27 }
```

〈그림 33.12〉 **동작하는 최종 해법**

팁: 조건에 while 문을 사용하자(if 문은 아니다)

멀티 쓰레드 프로그램에서 조건을 검사할 때에는 항상 **while** 문을 사용하는 것이 옳다. 시그널 전달의 의미에 따라 **if** 문을 사용하는 것은 맞을 수도 있을 뿐이다. 그러므로 항상 **while** 문을 사용하자, 그러면 작성한 코드가 의도한 대로 동작할 것이다.

조건 검사에 **while** 문을 사용하는 것은 거짓으로 깨운 경우(**spurious wakeup**)에 대처할 수 있도록 해 준다. 어떤 쓰레드 패키지는 구현상의 문제로 하나의 시그널에 의해서 두 개의 쓰레드가 깨어나는 경우도 가능하다 [L11]. 쓰레드가 조건을 재검사해야 하는 이유는 거짓으로 깨운 경우가 있기 때문이다.

그들이 직면했던 문제는 멀티 쓰레드 기반 메모리 할당 라이브러리의 예제로 가장 잘 설명될 수 있다. 그림 33.13에 이 이슈를 나타내는 예제를 보였다.

코드에서와 같이, 메모리 할당 코드를 호출하면 공간이 생길 때까지 기다려야할 수 있다. 반대로, 쓰레드가 메모리 반납시, 사용 가능한 메모리 공간의 발생을 알리는 시그널을 생성할 수 있다. 하지만 이 코드에는 문제가 있다. 어떤 쓰레드가(하나 이상의 쓰레드가 대기 중일 수 있으므로) 깨어나야 하는가?

이러한 시나리오를 생각해 보자. 빈 공간이 없다고 가정하고 쓰레드 T_a 가 **allocate(100)**을 실행하고 다음으로 쓰레드 T_b가 **allocate(10)**을 호출한다. 쓰레드 T_a와 T_b는 대기 상태에 들어간다. 지금은 그들의 요청을 만족시킬 수가 없다.

이 시점에서 쓰레드 T_c가 세 번째로 **free(50)**을 호출한다. 불행하게도 이 호출의 결과로 잘못된 쓰레드가 깨어날 수 있다. 10 Byte 공간을 필요로 하는 쓰레드 T_b가

```
1  // 몇 byte나 힙이 비어 있는가?
2  int bytesLeft = MAX_HEAP_SIZE;
3
4  // 락과 컨디션 변수가 필요함
5  cond_t c;
6  mutex_t m;
7
8  void *
9  allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...;          // 힙에서 메모리를 할당 받음
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // 시그널 전달 대상은?
23     Pthread_mutex_unlock(&m);
24 }
```

〈그림 33.13〉 **포함 조건: 예시**

깨어나야 한다. T_a는 깨어나면 안 된다. 때문에 이 코드는 제대로 동작하지 않는다. 그 이유는 어떤 쓰레드를 깨워야 할지 모르기 때문이다.

Lampson과 Redell이 제시한 해법은 단순하다. `pthread_cond_signal()`을 대기 중인 모든 쓰레드를 깨우는 `pthread_cond_broadcast()`로 바꿔서 사용하면 된다. 그렇게 함으로써 깨어나야 할 쓰레드가 있다면 깨어날 수 있도록 한다. 단점이라면 대기 중인 아직은 깨어나면 안 되는 여러 쓰레드가 불필요하게 깨어날 수도 있다는 점이 물론 성능에 안 좋은 영향을 미칠 수 있다. 그렇게 깨어난 쓰레드들은 깨어나서 조건을 재검사하고, 즉시 대기 상태로 다시 들어간다.

Lampson과 Redell은 이런 경우를 **포함 조건**(**covering condition**)이라고 했다. 왜냐하면 (보수적으로) 쓰레드가 깨어나야 하는 모든 경우를 다 포함하기 때문이다. 불필요하게 많은 쓰레드가 깨어나는 단점이 있다. 문맥 전환 오버헤드가 크다. 예리한 독자라면 이 방법을 앞에서 사용했을 수도 있다는 것을 알 것이다(컨디션 변수를 하나만 사용하는 생산자/소비자 문제를 보자). 하지만 그 경우에는 더 좋은 해법이 있었기 때문에 그 방법을 택했었다. 일반적으로 시그널을 브로드캐스트(broadcast)로 바꿨을 때만 프로그램이 동작한다면 아마도 버그가 존재하는 것일 거다. 앞서 다룬 메모리 할당 문제의 경우 브로드캐스트를 적용하는 것이 가장 자명한 해법이다.

33.4 요약

락 이상으로 중요한 동기화 기법인 컨디션 변수를 소개하였다. 프로그램 상태가 기대한 것과 다를 경우 쓰레드가 대기하도록 하여 컨디션 변수는 주요 동기화 문제 몇 가지를 깔끔하게 해결할 수 있다. 그 해법에는 그 유명한 생산자/소비자 그리고 포함 조건 문제도 포함한다. 좀 더 드라마틱하게 마치자 "그는 빅 브라더를 사랑했다." [Orw49].

참고 문헌

[BW 80] **"Experience with Processes and Monitors in Mesa"**
D.R. Redell B.W. Lampson
Communications of the ACM, 23:2, pages 105-117, February 1980
실제 시스템에서 시그널과 컨디션 변수 구현을 어떻게 할 것인가를 다룬 엄청난 논문이다. 깨어난다는 것이 어떤 의미를 갖는지를 알려주는 용어인 "Mesa" 시맨틱을 다룬다. 이전에 Tony Hoare가 만들었던 시맨틱은 그 이후에 'Hoare" 시맨틱으로 불리게 되었다. 이름의 어감 상 교실에서 평안한 얼굴로는 큰소리를 부를 수 없는 이름이다.

[Dij68] **"Cooperating sequential processes"**
E.W. Dijkstra
***http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF**, 1968*
Dijkstra가 남긴 또 다른 고전이다. 병행성에 대한 그의 초기작들을 읽으면 무엇을 알아야 할지 배우게 된다.

[Dij72] **"Information Streams Sharing a Finite Buffer"**
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
URL: `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF`
생산자/소비자 문제를 소개한 유명한 논문

[Dij01] **"My recollections of operating system design"**
E.W. Dijkstra
April, 2001
URL: `http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF`
우리 분야의 선구자들이 "인터럽트" 그리고 "스택"과 같은 기본적이고 근본적인 개념을 어떻게 만들게 되었는지가 궁금한 독자라면 이 글이 매우 흥미로울 것이다.

[Hoa74] **"Monitors: An Operating System Structuring Concept"**
C.A.R. Hoare
Communications of the ACM, 17:10, pages 549.557, October 1974
Hoare는 병행성에 대한 이론적 연구를 많이 진행했었다. 하지만 최소한 우리에게는 세상에서 가장 멋진 정렬 알고리즘인 퀵소트를 개발한 사람으로 가장 잘 알려져 있다.

[L11] **"Pthread cond signal Man Page"**
***http://linux.die.net/man/3/pthread_cond_signal**, March, 2011*
거짓으로 깨운 경우가 시그널 전달/깨우기 코드 내에서의 경쟁 조건 때문에 발생할 수 있다는 것을 설명하는 단순한 예제를 Linux 매뉴얼에서 보여준다.

[Orw49] **"1984"**
George Orwell
Secker and Warburg,1949
약간 가혹하기는 하지만 당연히 꼭 읽어야 하는 책이다. 말했으니 말인데, 책의 마지막 문장을 인용해서 결말을 알려준 것 같다. 미안하다! 만약 정부가 이 책을 읽는다면 정부는 "배로 더 좋아요."라고 생각한다고 말해주고 싶다. NSA에서 일하는 친구들 듣고 있나?