30

막간: 쓰레드 API

이번 장에서는 쓰레드 API의 주요 부분을 간략하게 다룬다. 각 부분에 대한 자세한 내용은 이후의 장에서 다루도록 한다. 더 상세한 내용은 여러 책과 온라인 자료 [Bir89; But97; BFN96; KSS96]에서 얻을 수 있다. 이후의 장들에서 컨디션 변수와 락에 대해 예제와 함께 천천히 소개할 것이다. 이번 장은 참고용으로 사용하면 좋겠다.

핵심 질문: 쓰레드를 생성하고 제어하는 방법
운영체제가 쓰레드를 생성하고 제어하는 데 어떤 인터페이스를 제공해야 할까? 어떻게 이 인터페이스를 설계해야 쉽고 유용하게 사용할 수 있을까?

30.1 쓰레드 생성

멀티 쓰레드 프로그램을 작성 시, 가장 먼저할 일은 새로운 쓰레드의 생성이다. 쓰레드 생성을 위해서는 해당 인터페이스가 존재해야 한다. POSIX에서는 쉽게 할 수 있다.

```
1 #include <pthread.h>
2 int
3 pthread_create(      pthread_t *       thread,
4                 const pthread_attr_t *  attr,
5                       void *            (*start_routine)(void*),
6                       void *            arg);
```

복잡하게 보이지만(특히, 함수 포인터를 사용해 보지 않았다면 더욱), 실제로는 그리 어렵지 않다. `thread`, `attr`, `start_routine` 그리고 `arg`라는 4개의 인자가 있다. 먼저 `thread`는 `pthread_t` 타입 구조체를 가리키는 포인터이다. 이 구조가 쓰레드와 상호작용하는 데 사용되기 때문에 쓰레드 초기화 시 `pthread_create()`에 이 구조체를 전달한다.

두 번째 인자 `attr`은 쓰레드의 속성을 지정하는 데 사용한다. 스택의 크기와 쓰레드의 스케줄링 우선순위 같은 정보를 지정하기 위해서 사용될 수 있다. 개별 속성은 `pthread_attr_init()` 함수를 호출하여 초기화한다. 구체적인 내용은 매뉴얼을 참고하기 바란다. 대부분의 경우에 디폴트 값을 지정하면 충분하다. 간단히 `NULL`을 전달하면 된다.

세 번째 인자는 인자들 중에 가장 복잡해 보인다. 이 쓰레드가 실행할 함수를 나타낸다. C 언어에서는 이를 함수 포인터라고 부르고, 다음과 같은 정보가 필요하다고 알려준다. 이 함수는 `void *` 타입의 인자 한 개를 전달받는다(함수 이름 `start_routine` 다음에 괄호 안에 표시된 것처럼). 그리고 `void *` (즉, `void` 포인터) 타입의 값을 반환한다.

`void` 포인터 타입 대신 `integer`를 인자로 사용하는 루틴이라면 선언은 다음과 같이 될 것이다.

```
int pthread_create(...,  // 처음 두 인자는 동일함
                   void *   (*start_routine)(int),
                   void *   arg);
```

`void` 포인터 타입을 인자로 받지만, `integer` 타입을 반환한다면 다음과 같은 형식이 될 것이다.

```
int pthread_create(...,  // 처음 두 인자는 동일함
                   int      (*start_routine)(void*),
                   void *   arg);
```

마지막으로 네 번째 인자인 `arg`는 실행할 함수에게 전달할 인자를 나타낸다. 왜 `void` 포인터 타입이 필요하지라고 질문할 수도 있다. 그 이유는 간단하다. `void` 포인터를 `start_routine` 함수의 인자로 사용하면, *어떤* 데이터 타입도 인자로 전달할 수 있고, 반환 값의 타입으로 사용하면 쓰레드는 *어떤* 타입의 결과도 반환할 수 있다.

그림 30.1의 예제를 살펴보자. 두 개의 인자를 전달받는 새로운 쓰레드를 생성하는데 두 인자는 우리가 정의한 `myarg_t` 타입으로 묶여진다. 쓰레드가 생성되면 전달받은 인자의 타입을 예상하고 있는 타입으로 변환할 수 있고, 원하는 대로 인자를 풀어낼 수 있다.

```
1  #include <pthread.h>
2
3  typedef struct __myarg_t {
4      int a;
5      int b;
6  } myarg_t;
7
8  void *mythread(void *arg) {
9      myarg_t *m = (myarg_t *) arg;
10     printf("%d %d\n", m->a, m->b);
11     return NULL;
12 }
13
14 int
15 main(int argc, char *argv[]) {
16     pthread_t p;
17     int rc;
18
19     myarg_t args;
20     args.a = 10;
21     args.b = 20;
22     rc = pthread_create(&p, NULL, mythread, &args);
23     ...
24 }
```

〈그림 30.1〉 **쓰레드 생성**

자, 끝났다. 쓰레드를 생성하고 나면, 자신만의 스택까지 완벽히 갖춘, 그리고 프로그램의 실행 중인 모든 쓰레드와 같은 주소 공간에서 실행되는 또 하나의 실행 개체를 보유하게 된다. 이제부터가 진짜 재미 있는 부분이니 기대하길!

30.2 쓰레드 종료

위의 예제는 쓰레드를 생성하는 법을 보았다. 다른 쓰레드가 작업을 완료할 때까지 기다려야 한다면 어떻게 해야 할까? 다른 쓰레드의 완료를 기다리기 위해서는 뭔가 특별한 조치를 해야 한다. POSIX 쓰레드에서는 `pthread_join()`을 부르는 것이다.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

이 루틴은 두 개의 인자를 받는다. 첫 번째 `pthread_t` 타입 인자는 어떤 쓰레드를 기다리려고 하는지 명시한다. 이 변수는 쓰레드 생성 루틴에 의해 초기화된다(자료구조에 대한 포인터를 `pthread_create()`의 인자로 전달하여). 이 구조체를 보관해 놓으면, 그 쓰레드가 끝나기를 기다릴 때 사용할 수 있다.

두 번째 인자는 반환 값에 대한 포인터이다. 루틴이 임의의 데이터 타입을 반환할 수 있기 때문에 void에 대한 포인터 타입으로 정의한다. `pthread_join()` 루틴은 전달된 인자의 값을 *변경하기* 때문에 값을 전달하는 것이 아니라 그 값에 대한 포인터를 전달해야 한다.

그림 30.2에 있는 또 다른 예제를 살펴보자. 이 코드에서는 이전 예제와 마찬가지로 쓰레드를 생성하여 `myarg_t` 데이터 타입을 통하여 2개의 인자를 전달했다. 반환 값으로 `myret_t` 타입이 사용된다. 메인 쓰레드는 `pthread_join()` 루틴[1] 안에서 기다리는 중이다. 쓰레드가 실행을 마치면 메인 쓰레드가 리턴하고, 종료한 쓰레드가 반환한 값, 즉 `myret_t` 안에 들어 있는 무슨 값이든 접근할 수 있게 된다.

이 예제에서 주목할 만한 몇 가지가 있다. 먼저 여러 인자를 한 번에 전달하기 위해 묶고 해체하는 불편한 과정을 항상 해야 하는 것은 아니다. 예를 들면 인자가 없는 쓰레드를 생성할 때에는 NULL을 전달하여 쓰레드를 생성할 수도 있다. 비슷한 예로 반환 값이 필요 없다면 `pthread_join()`에 NULL을 전달할 수도 있다.

두 번째는 값 하나만(예, int 값) 전달해야 한다면 인자를 전달하기 위해 묶을 필요가 없다. 그림 30.3에 그 예가 있다. 이 경우에 인자와 반환 값을 구조체로 묶을 필요가 없기 때문에 간단하게 할 수 있다.

세 번째로 유의해야 할 것은 쓰레드에서 값이 어떻게 반환되는지에 대해 각별한 신경을 써야 한다는 것이다. 특히, 쓰레드의 콜 스택에 할당된 값을 가리키는 포인터를 반환하지 마라. 이런 포인터를 반환한다면, 어떤 일이 발생할 것 같은가?(생각해 보라!) 여기에 그림 30.2에서 사용한 예를 변형하여 만든 매우 위험한 코드가 있다.

1) 이 예제에서 래퍼 함수를 사용하는 것에 유의해야 한다. 우리는 래퍼 함수인 `Malloc()`, `Pthread_join()`, `Pthread_create()`를 호출한다. 이 래퍼 함수는 첫 문자가 소문자인 원래의 API를 호출하고, 이 루틴들이 예상치 못한 값을 반환하지 않도록 보장한다.

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

〈그림 30.2〉 **쓰레드 종료 기다리기**

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // 스택에 할당했다. 이러면 안 된다!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

이 코드에서 `mythread`의 스택에 변수 `r`이 할당되어 있다. 이 값은 쓰레드가 리턴할 때 자동적으로 해제된다(어쨌건, 이게 스택이 사용하기 쉬운 이유이다). 현재 해제된 변수를 가리키는 포인터를 반환하는 것은 온갖 종류의 좋지 않은 결과를 가져온다. 분명 반환받았다고 생각하는 값을 출력해 보면 (꼭 그런 건 아니지만) 놀라게 될 것이다. 직접 결과를 확인해 보라[2].

마지막으로, `pthread_create()`를 사용하여 쓰레드를 생성하고 직후에 `pthread_join()`을 호출한다는 것은 쓰레드를 생성하는 아주 이상한 방법이다. 사실 작업을 똑같이 할

2) 다행스럽게 당신이 이러한 코드를 작성하면 gcc 컴파일러가 투덜댈 것이다. 컴파일러의 경고 메시지를 잘 읽어봐야 하는 또 다른 이유가 되겠다.

```
1  void *mythread(void *arg) {
2      int m = (int) arg;
3      printf("%d\n", m);
4      return (void *) (arg + 1);
5  }
6  int main(int argc, char *argv[]) {
7      pthread_t p;
8      int rc, m;
9      Pthread_create(&p, NULL, mythread, (void *) 100);
10     Pthread_join(p, (void **) &m);
11     printf("returned %d\n", m);
12     return 0;
13 }
```

〈그림 30.3〉 **쓰레드에 간단한 인자 전달하기**

수 있는 더 쉬운 방법이 있다. 이를 **프로시저 호출**(**procedure call**)이라고 부른다. 여러 개의 쓰레드를 생성해 놓고 쓰레드가 끝나기를 기다리는 게 보통일 것이다. 그렇지 않다면 쓰레드를 사용할 이유가 없을 테니 말이다.

모든 멀티 쓰레드 코드가 조인 루틴을 사용하지는 않는다. 예를 들면 멀티 쓰레드 웹서버의 경우 여러 개의 작업자 쓰레드를 생성하고 메인 쓰레드를 이용하여 사용자 요청을 받아 작업자에게 전달하는 작업을 무한히 할 것이다. 이런 프로그램은 `join`을 사용할 필요가 없다. 하지만, 특정 작업을 병렬적으로 실행하기 위해 쓰레드를 생성하는 병렬 프로그램의 경우에는 종료 전 혹은 계산의 다음 단계로 넘어가기 전에 병렬 수행 작업이 모두 완료되었다는 것을 확인하기 위해 `join`을 사용한다.

30.3 락

쓰레드의 생성과 조인 다음으로 POSIX 쓰레드 라이브러리가 제공하는 가장 유용한 함수는 **락**(**lock**)을 통한 임계 영역에 대한 상호 배제 기법이다. 이러한 목적을 위하여 사용되는 가장 기본적인 루틴은 다음과 같이 쌍으로 이루어져 있다.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

이 루틴은 이해하고 사용하기 쉬워야 한다. 그래야 임계 영역을 원하는 방식으로 동작하도록 보호할 때 사용할 수 있다. 다음과 같은 코드를 예상할 수 있다.

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // 또는 다른 임계 영역의 코드를 사용할 수 있음
pthread_mutex_unlock(&lock);
```

이 코드가 하고자 하는 바는 다음과 같다. `pthread_mutex_lock()`가 호출되었을 때 다른 어떤 쓰레드도 락을 가지고 있지 않다면 이 쓰레드가 락을 얻어 임계 영역에 진입한다. 만약 다른 쓰레드가 락을 가지고 있다면, 락 획득을 시도하는 쓰레드는 락을 얻을 때까지 호출에서 리턴하지 않는다(리턴했다면 락을 가지고 있던 쓰레드가 언락(unlonk)을 호출하여 락을 양도했다는 것을 의미한다). 많은 쓰레드들이 락 획득 함수에서 대기중 일 수 있다. 락을 획득한 쓰레드만이 언락을 호출해야 한다.

안타깝게도 이 코드는 두 가지 측면에서 올바르게 동작하지 않는다. 첫 번째, 초기화를 하지 않았다. 올바른 값을 가지고 시작했다는 것을 보장하기 위해 모든 락은 올바르게 초기화되어야 한다. 그래야 락과 언락을 호출하였을 때 의도한 대로 동작할 수 있다.

POSIX 쓰레드를 사용할 때 락을 초기화하는 방법은 두 가지이다. 한 가지 방법은 다음과 같이 `PTHREAD_MUTEX_INITIALIZER`를 사용하는 것이다.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

위 연산은 락을 디폴트 값으로 설정한다. 동적으로 초기화하는 방법은(즉, 실행 중에) 다음과 같이 `pthread_mutex_init()`을 호출하는 것이다.

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // 성공했는지 꼭 확인해야 한다!
```

이 루틴의 첫 번째 인자는 락 자체의 주소이고, 반면에 두 번째 인자는 선택 가능한 속성이다. 이 속성에 대해서는 스스로 찾아 읽어보기 바란다. NULL을 전달하면 디폴트 값을 사용한다. 정적, 동적 두 방법 모두 사용할 수 있지만 우리는 후자인 동적 방법을 주로 사용한다. 락 사용이 끝났다면 초기화 API와 상응하는 `pthread_mutex_destroy()`도 호출해야 한다는 것을 주의하라. 이 부분도 매뉴얼을 참고하기 바란다.

위 코드의 두 번째 문제는 락과 언락을 호출할 때 에러 코드를 확인하지 않는다는 것이다. UNIX 시스템에서 호출하는 거의 모든 라이브러리 루틴과 마찬가지로 이 루틴들도 실패할 수 있다! 당신의 코드가 제대로 에러 코드를 검사하지 않는다면, 코드는 조용히 실패하게 된다. 이 경우 여러 쓰레드가 동시에 임계 영역에 들어갈 수 있다. 최소한 래퍼 함수를 사용하여 해당 루틴이 성공적으로 처리되었는지 확인해야 한다(예, 그림 30.4). 실제 복잡한 프로그램에서, 락이나 언락이 실패한 경우 그냥 종료하면 안되고 적절한 대응을 해야 한다.

```
1 // 이 방법을 써서 코드를 깔끔하게 유지하되 오류가 없는지 확인해야 한다.
2 // 프로그램이 오류에도 문제가 없었을 때에만 사용하자.
3 void Pthread_mutex_lock(pthread_mutex_t *mutex) {
4     int rc = pthread_mutex_lock(mutex);
5     assert(rc == 0);
6 }
```

〈그림 30.4〉 **래퍼 함수의 예시**

락과 언락 루틴 외에 `pthread` 라이브러리에서 락 관련 루틴들이 더 존재한다. 특히 관심을 가질 만한 루틴이 2개 더 있다.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

이 두 함수는 락을 획득하는 데 사용된다. `trylock` 버전은 락이 이미 사용 중이라면 실패 코드를 반환한다. `timedlock`은 타임아웃이 끝나거나 락을 획득하거나의 두 조건 중 하나가 발생하면 리턴한다. `timedlock`의 타임아웃을 0으로 설정하면 `trylock`과 동일하게 동작한다. 이 두 함수는 사용하지 않는 것이 좋다. 그러나 락 획득 루틴에서

무한정 대기하는 상황을 피하기 위해 사용되기도 한다. 이러한 경우를 앞으로 (예, 교착 상태를 공부할 때) 보게 될 것이다.

30.4 컨디션 변수

쓰레드 라이브러리가 POSIX 쓰레드의 경우에는 확실히, 제공하는 주요한 구성 요소로 **컨디션 변수**(**condition variable**)가 있다. 한 쓰레드가 계속 진행하기 전에 다른 쓰레드가 무언가를 해야 쓰레드 간에 일종의 시그널 교환 메커니즘이 필요하다. 이런 경우 컨디션 변수가 사용된다. 두 개의 기본 루틴은 다음과 같다.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

컨디션 변수 사용을 위해서는 이 컨디션 변수와 연결된 락이 "반드시" 존재해야 한다[3]. 위의 루틴 중 하나를 호출하기 위해서는 그 락을 갖고 있어야 한다.

첫 번째 루틴 `pthread_cond_wait()`는 호출 쓰레드를 수면(sleep) 상태로 만들고 다른 쓰레드로부터의 시그널을 대기한다. 현재 수면 중인 쓰레드가 관심 있는 무언가가 변경되면 시그널을 보낸다. 전형적인 용례는 다음과 같다.

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3
4 Pthread_mutex_lock(&lock);
5 while (ready == 0)
6     Pthread_cond_wait(&cond, &lock);
7 Pthread_mutex_unlock(&lock);
```

이 코드에서는 연관된 락과 컨디션 변수[4]를 초기화한 후에 쓰레드는 `ready` 변수가 0인지 검사한다. `ready` 변수 값이 0이라면, 다른 쓰레드가 깨워줄 때까지 잠들기 위해 대기 루틴을 호출한다.

다른 쓰레드에서 실행될 잠자는 쓰레드를 깨우는 코드는 다음과 같다.

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

이 코드에서 유의할 몇 가지가 있다. 첫 번째, 시그널을 보내고 전역 변수 `ready`를 수정할 때 반드시 락을 가지고 있어야 한다. 이를 통해 경쟁 조건이 발생하지 않는다는 것을 보장한다.

두 번째, 시그널 대기 함수에서는 락을 두 번째 인자로 받고 있지만, 시그널 보내기 함수에서는 조건만을 인자로 받는 것에 유의해야 한다. 이런 차이의 이유는 시그널 대기 함수는 호출 쓰레드를 재우는 것 외에 락도 *반납(release) 해야* 하기 때문이다. 락을 반납하지 않았다고 상상해 보자. 어떻게 다른 쓰레드가 락을 얻어 이 쓰레드를 깨우기

3) 역자 주: 꼭, 꼭 갖고 있어야 한다.

4) 정적 초기화 루틴 `PTHREAD_COND_INITIALIZER` 대신 `pthread_cond_init()`(그리고 상응하는 `pthread_cond_destroy()`)를 사용할 수 있다. 더 많은 일을 해야 할 것 같지? 실제로 그렇다.

위한 시그널을 보낼 수 있겠는가? `pthread_cond_wait()`는 깨어나서 리턴하기 *직전에* 락을 다시 획득한다. 처음 락을 획득한 때부터 마지막에 락을 반납할 때까지 `pthread_cond_wait()`를 실행한 쓰레드들은 항상 락을 획득한 상태로 실행된다는 것을 보장한다.

마지막으로 *매우* 중요한 특이 사항이 있다. 대기하는 쓰레드가 조건을 검사할 때 `if` 문을 사용하는 대신 `while` 문을 사용한다는 것이다. 이 주제는 컨디션 변수를 다루는 장에서 자세히 다룰 것이다. `while` 문을 사용하는 것이 일반적으로 간단하고 안전하다. `pthread` 라이브러리에서(실수로 또는 부주의하게) 변수를 제대로 갱신하지 않고 대기하던 쓰레드를 깨울 수 있다. 이런 경우 재검사를 하지 않는다면 대기하던 쓰레드는 조건이 변경되지 않았더라도 변경되었다고 생각할 것이다. 때문에 시그널의 도착은 변경 사실을 알리는 것이 아니라, *변경된 것 같으니 검사해보라*는 정도의 힌트로 간주하는 것이 더 안전하다.

두 쓰레드 간에 시그널을 주고 받아야 할 때, 락과 컨디션 변수를 사용하는 대신 간단한 플래그를 사용하여 구현하고 싶을 것이다. 예를 들어 앞에서의 대기 코드를 다음과 같이 다시 코딩할 수 있다.

```
while (ready == 0)
    ; // 회전
```

이 코드에 상응하는 시그널 보내기 코드는 다음과 같을 것이다.

```
ready = 1;
```

절대로 하지 마라. 이유는 다음과 같다. 첫째, 많은 경우에 이 코드는 성능이 좋지 않다. 조건 검사를 위해 오랫동안 반복문을 실행하여 검사하는 것은 CPU 사이클의 낭비를 초래한다. 둘째, 오류가 발생하기 쉽다. 최근 연구 [Xio+10]에 따르면, 쓰레드 간에 동기화를 하기 위해 플래그를 사용할 때(위의 경우처럼) 놀랄 정도로 실수하기 쉽다. 그 연구에 의하면 *임시방편적인* 동기화 방법을 사용한 경우, 대략 절반 정도가 버그를 유발하였다! 불편하다고 피하지 말아라. 컨디션 변수를 사용하지 않고도 할 수 있다고 생각하더라도, 꼭 컨디션 변수를 사용하기 바란다.

컨디션 변수가 아직 명확히 이해가 안되더라도 (아직은) 너무 걱정하지 마라—이후의 장에서 더 자세하게 살펴볼 것이다. 그때까지는 컨디션 변수가 있다는 것과 언제 어떻게 사용한다는 정도만 알고 있으면 충분하다.

30.5 컴파일과 실행

이 장에서 사용한 예제 코드들은 상대적으로 작동시키기 쉽다. 예제들을 컴파일 하기 위해서는 `pthread.h` 헤더를 포함시켜야 한다. `-pthread` 플래그를 명령어 링크 옵션 부분에 추가하여 사용하여 pthread 라이브러리와 링크할 수 있도록 명시해야 한다.

예를 들어 간단한 멀티 쓰레드 프로그램을 컴파일하기 위하여 당신이 해야 할 일은 다음 명령어를 입력하는 것 뿐이다.

```
prompt> gcc -o main main.c -Wall -pthread
```

`main.c` 가 `pthread` 헤더를 포함하고 있는 한 병행 프로그램이 성공적으로 컴파일 된다. 프로그램의 동작 여부는 늘 그렇듯 전혀 다른 문제이다.

30.6 요약

쓰레드 생성과 락을 통한 상호 배제의 구현, 컨디션 변수를 이용한 시그널과 대기 등 `pthread` 라이브러리의 기본 지식을 소개하였다. 강인하고 효율적인 멀티 쓰레드 코드를 작성하기 위해서는 인내와 세심한 주의가 답이다.

멀티 쓰레드 코드를 작성하는 데 도움될 만한 몇 가지 팁과 함께 이 장을 마치고자 한다(여담에 나와 있는 내용을 참고하기 바란다). API에는 흥미로운 내용들이 더 많다. 좀 더 많은 정보를 원한다면 Linux 시스템에서 `man -k pthread`를 실행시켜 백여 개가 넘는 API들의 인터페이스를 확인해 보기 바란다. 하지만 이 장에서 다룬 기본적인 논의로도 정교한 (그리고 올바르고 성능 좋은) 멀티 쓰레드 프로그램을 작성할 수 있어야 한다. 쓰레드에 있어 어려운 부분은 API가 아니라 병행 프로그램을 구현하는 정교한 사고 방식이다. 더 많은 걸 배우기 위해 계속 읽어 나가기 바란다.

여담: 쓰레드 API 의 지침

POSIX 쓰레드 라이브러리(또는 다른 어떤 쓰레드 라이브러리)를 사용하여 멀티 쓰레드 프로그램을 만드는 데 있어 기억해야 할 중요한 몇 가지가 있다.

- **간단하게 작성하라**. 무엇보다도 락을 획득하거나 쓰레드끼리 시그널을 주고 받는 코드는 가능한 한 간단해야 한다. 쓰레드 간의 복잡한 상호 동작은 버그를 만든다.
- **쓰레드 간의 상호 동작을 최소로 하라**. 쓰레드끼리 상호 작용하는 방법의 개수를 최소로 해야 한다. 각 상호 작용은 깊게 생각한 후에 검증된 방법을 사용하여 구현되어야 한다(앞으로 다룰 장에서 검증된 방법에 대해서 배우게 될 것이다).
- **락과 컨디션 변수를 초기화하라**. 초기화하지 않고 사용하면 어떤 때는 동작하지만 때로는 매우 이상한 방식으로 실패할 수 있다.
- **반환 코드를 확인하라**. 물론 C와 Unix 프로그램을 작성할 때에는 언제나 반환 코드를 확인해야 한다. 쓰레드를 사용할 때도 역시 마찬가지이다. 확인하지 않을 경우 기이하고 이해하기 어려운 동작을 초래하기 때문에 (a) 소리를 지르게 만들거나 (b) 머리카락을 쥐어뜯거나 (c) 둘 다하게 만들 것이다.
- **쓰레드 간에 인자를 전달하고 반환받을 때는 조심해야 한다**. 특히, 스택에 할당된 변수에 대한 참조를 전달할 경우 뭔가 잘못하고 있는 것이 맞을 것이다.
- **각 쓰레드는 개별적인 스택을 가진다**. 실행 중인 어떤 함수 내에 지역적으로 할당된 변수가 있다면 그 변수는 본질적으로 쓰레드 전용으로 사용되어야 하고 다른 쓰레드가 (쉽게) 접근할 수 없어야 한다. 쓰레드 간에 데이터를 공유하려면 힙에 할당하거나, 전역적으로 접근이 가능한 위치에 있어야 한다.
- **쓰레드 간에 시그널을 보내기 위해 항상 컨디션 변수를 사용하라**. 간단한 플래그를 쓰고 싶은 유혹이 들 수도 있겠지만 하지 마라.
- **매뉴얼을 사용하라**. 특히 Linux에 `pthread`에 대한 설명에는 많은 정보가 있다. 여기서 다룬 대부분을 설명하며 어떤 내용은 더 자세하게 다루고 있다. 꼼꼼하게 읽어 보기 바란다.

참고 문헌

[Bir89] **"An Introduction to Programming with Threads"**
Andrew D. Birrell
DEC Technical Report, January, 1989
URL: `https://birrell.org/andrew/papers/035-Threads.pdf`
쓰레드 프로그래밍에 관한 고전이지만 오래된 소개. 여전히 읽을 만한 가치가 있고 무료로 얻을 수 있다.

[But97] **"Programming with POSIX Threads"**
David R. Butenhof
Addison-Wesley, May 1997
쓰레드에 관한 또 다른 책

[BFN96] **"PThreads Programming: A POSIX Standard for Better Multiprocessing"**
Dick Buttlar, Jacqueline Farrell, and Bradford Nichols

O'Reilly, September 1996
훌륭하고 실용적인 출판사인 O'Reilly에서 출간한 괜찮은 책이다. Perl, Python 그리고 Javascript(특히, Crockford가 쓴 "Javascript: The Good Parts")에 관한 책들을 포함하여 이 출판사에서 출간한 많은 책들이 책장에 꽂혀 있을 것이다.

[KSS96] **"Programming With Threads"**
Steve Kleiman, Devang Shah, and Bart Smaalders
Prentice Hall, January 1996
이 분야의 아마도 가장 좋은 책 중에 하나일 것이다. 지역 도서관에서 구하든지 어머님의 것을 훔쳐라. 진지하게 말하자면, 어머님께 빌려달라고 하자. 빌려주실 테니 걱정하지 말자.

[Xio+10] **"Ad Hoc Synchronization Considered Harmful"**
Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma
OSDI 2010, Vancouver, Canada
이 논문은 간단하게 보이는 동기화 코드가 얼마나 많은 버그를 만들어내는지 보여 준다. 컨디션 변수를 사용하는 것과 시그널 전달을 제대로 사용하자.