29

병행성 : 개요

지금까지 운영체제가 다루는 기본 개념들의 발전 과정을 살펴보았다. 하나의 물리적 CPU를 다수의 **가상 CPU**로 확장하여 마치 여러 개의 프로그램이 동시에 실행하는 듯한 착시를 만들었다. 그리고 개별적인 프로세스가 모두 독립적으로 많은 **가상 메모리**를 가지는 것처럼 보이게 만들었다. **주소 공간**(**address space**)이라는 개념을 통해 각 프로그램들이 마치 자신만의 메모리를 가지고 있는 것처럼 동작하는데, 사실은 운영체제가 물리 메모리를 여러 개의 주소 공간이 서로 번갈아 가면서 사용하게 한다.

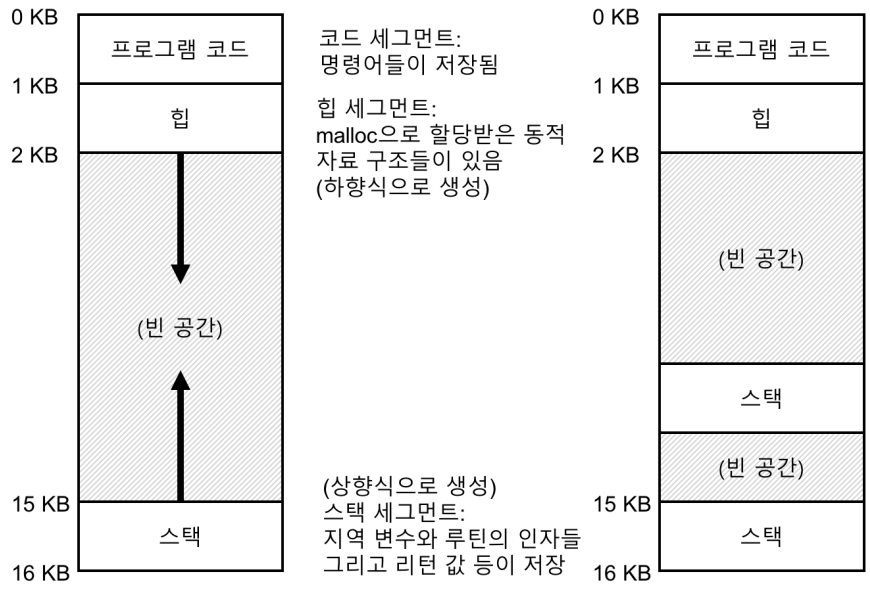
이번 장에서는 프로세스를 위한 새로운 개념인 **쓰레드**(**thread**)를 소개한다. 프로그램에서 한 순간에 하나의 명령어만을 실행하는(단일 PC[1] 값) 고전적인 관점에서 벗어나 멀티 쓰레드 프로그램은 하나 이상의 실행 지점(독립적으로 불러 들여지고 실행될 수 있는 여러 개의 PC 값)을 가지고 있다. 멀티 쓰레드를 이해하는 다른 방법은 각 쓰레드가 프로세스와 매우 유사하지만, 차이가 있다면 쓰레드들은 주소 공간을 공유하기 때문에 동일한 값에 접근할 수 있다는 것이다.

하나의 쓰레드의 상태는 프로세스의 상태와 매우 유사하다. 쓰레드는 어디서 명령어들을 불러 들일지 추적하는 프로그램 카운터(PC)와 연산을 위한 레지스터들을 가지고 있다. 만약 두 개의 쓰레드가 하나의 프로세서에서 실행 중이라면 실행하고자 하는 쓰레드 (T2)는 반드시 **문맥 교환**(**context switch**)을 통해서 실행 중인 쓰레드 (T1)와 교체되어야 한다. 쓰레드 간의 문맥 교환은 T1이 사용하던 레지스터들을 저장하고 T2가 사용하던 레지스터의 내용으로 복원한다는 점에서 프로세스의 문맥 교환과 유사하다. 프로세스가 문맥 교환을 할 때에 프로세스의 상태를 **프로세스 제어 블럭**(**process control block**, **PCB**)에 저장하듯이 프로세스의 쓰레드들의 상태를 저장하기 위해서는 하나 또는 그 이상의 **쓰레드 제어 블럭**(**thread control block**, **TCB**)이 필요하다. 가장 큰 차이 중 하나는 프로세스의 경우와 달리 쓰레드 간의 문맥 교환에서는 주소 공간을 그대로 사용한다는 것이다(사용하고 있던 페이지 테이블을 그대로 사용하면 된다).

쓰레드와 프로세스의 또 다른 차이는 스택에 있다. 고전적 프로세스 주소 공간과 같은 간단한 모델(**단일 쓰레드** 프로세스)에서는 스택이 하나만 존재한다. (그림 29.1 좌측) 주로 주소 공간의 하부에 위치한다.

1) 역자 주: 제??장에서 나왔었다. PC는 Program Counter의 준말이다.

반면에 멀티 쓰레드 프로세스의 경우에는 각 쓰레드가 독립적으로 실행되며 쓰레드가 실행하기 위해 여러 루틴들을 호출할 수 있다. 주소 공간에는 하나의 스택이 아니라 쓰레드마다 스택이 할당되어 있다. 두 개의 쓰레드를 가지는 멀티 쓰레드 프로세스의 주소 공간은 단일 쓰레드 프로세스의 주소 공간과 다르다(그림 29.1 우측).



〈그림 29.1〉 **단일 쓰레드와 멀티 쓰레드의 주소 공간**

이 그림에서는 두 개의 스택이 프로세스 주소 공간에 존재하는 것을 볼 수 있다. 스택에서 할당되는 변수들이나 매개변수, 리턴 값, 그리고 그 외에 스택에 넣는 것들은 해당 쓰레드의 스택인 쓰레드-로컬 저장소(thread-local storage)라 불리는 곳에 저장된다.

쓰레드-로컬 저장소로 인해서 정교한 주소 공간의 배치가 무너진다는 것을 알았을 것이다. 전에는 스택과 힙이 독립적으로 확장되기 때문에 주소 공간에 더 이상 공간이 없는 경우에만 문제가 생겼었다. 이제는 상황이 예전처럼 깔끔하지 않다. 다행스러운 것은 스택의 크기가 아주 크지 않아도 되기 때문에 대부분의 경우에는 문제가 되지 않는다(재귀 호출을 아주 많이 하는 경우는 제외한다).

29.1 예제 : 쓰레드 생성

한 쓰레드는 "A"라고 출력하고 다른 쓰레드는 "B"라고 출력하는 독립적인 두 개의 쓰레드를 생성하는 프로그램을 실행시킨다고 해 보자. 코드는 그림 29.2에 나타나 있다.

메인 프로그램은 각각 `mythread()` 함수를 실행할 두 개의 쓰레드를 생성한다. 이때 각 `mythread()` 함수는 서로 다른 인자를 전달받는다. 스케줄러의 동작에 따라 다르겠지만, 쓰레드가 생성되면, 즉시 실행될 수도 있고, 준비(Ready) 상태에서

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A");
16     assert(rc &=& 0);
17     rc = pthread_create(&p2, NULL, mythread, "B");
18     assert(rc &=& 0);
19     // 종료 할 수 있도록 대기 중인 쓰레드 병합하기
20     rc = pthread_join(p1, NULL); assert(rc &=& 0);
21     rc = pthread_join(p2, NULL); assert(rc &=& 0);
22     printf("main: end\n");
23     return 0;
24 }
```

〈그림 29.2〉 **간단한 쓰레드 생성 코드 (t0.c)**

실행은 되지 않을 수 있다. 두 개의 쓰레드(T1과 T2)를 생성한 후에 메인 쓰레드는 `pthread_join()`을 호출하여 특정 쓰레드의 동작의 종료를 대기한다.

이 프로그램의 가능한 실행 순서를 살펴보도록 하자. 실행 순서도(그림 29.3)에서 시간은 아래 방향으로 진행하고 각 열은 각 쓰레드(메인, 쓰레드 1 또는 쓰레드2)가 언제 실행 중인지를 나타낸다.

Main	**쓰레드 1**	**쓰레드 2**
실행 시작		
"main: begin" 출력		
쓰레드 1 생성		
쓰레드 2 생성		
T1을 대기		
	실행	
	"A" 출력	
	리턴	
T2를 대기		
		실행
		"B" 출력
		리턴
"main: end" 출력		

〈그림 29.3〉 **쓰레드의 실행 추적 (1)**

Main	쓰레드 1	쓰레드 2
실행 시작		
“main: begin” 출력		
쓰레드 1 생성		
	실행	
	“A” 출력	
	리턴	
쓰레드 2 생성		
		실행
		“B” 출력
		리턴
T1을 대기		
즉시 리턴 ; T1 완료		
T2를 대기		
즉시 리턴 ; T2 완료		
“main: end” 출력		

〈그림 29.4〉 **쓰레드의 실행 추적 (2)**

유의할 점은 도표에서 나타내는 실행 순서가 쓰레드가 유일한 실행 가능 순서가 아니라는 것이다. 스케줄러가 특정 시점에 실행하는 쓰레드에 따라 다양한 순서가 있을 수 있다. 예를 들어 그림 29.4에서 나타낸 실행 순서도와 같이 쓰레드가 생성된 후 즉시 실행될 수도 있다.

쓰레드 1이 쓰레드 2보다 먼저 생성된 경우라 하더라도 만약 스케줄러가 쓰레드 2를 먼저 실행하면 “B”가 “A” 보다 먼저 출력될 수도 있다. 먼저 생성되었다고 해서 먼저 실행될 것이라는 가정을 할 어떤 이유도 없다. 그림 29.5는 쓰레드 2가 쓰레드 1보다 먼저 실행되는 마지막 경우를 보인다.

쓰레드의 생성이 함수의 호출과 유사하게 보인다. 함수 호출에서는 함수 실행 후에 호출자(caller)에게 리턴하는 반면에 쓰레드의 생성에서는 실행할 명령어들을 갖고 있는 새로운 쓰레드가 생성되고, 생성된 쓰레드는 호출자와는 별개로 실행된다. 쓰레드 생성 함수가 리턴되기 전에 쓰레드가 실행될 수도 있고, 그보다 이후에 실행될 수도 있다.

이 예제에서 알 수 있듯이, 쓰레드는 일을 복잡하게 만든다. 이 간단한 예제에서도 어떤 쓰레드가 언제 실행되는지 알기 어렵다. 컴퓨터는 병행성이라는 주제가 아니더라도 충분히 어려운데, 불행하게도 이 문제 때문에 더 어려워진다. 훨씬.

29.2 훨씬 더 어려운 이유 : 데이터의 공유

앞의 간단한 쓰레드 예제를 통해 쓰레드의 생성 방법을 살펴보았고, 실행 순서는 스케줄러의 동작에 따라 바뀔 수 있다는 것을 보았다. 예제에서 나타나지 않은 부분은 쓰레드가

Main	**쓰레드 1**	**쓰레드 2**
실행 시작		
"main: begin" 출력		
쓰레드 1 생성		
쓰레드 2 생성		
		실행
		"B" 출력
		리턴
T1을 대기		
	실행	
	"A" 출력	
	리턴	
T2를 대기		
즉시 리턴; T2 완료		
"main: end" 출력		

〈그림 29.5〉 **쓰레드의 실행 추적 (3)**

공유 데이터를 접근하기 위해 상호작용하는 과정이다.

그림 29.6에 나와 있는 코드를 사용하여 전역 공유 변수를 갱신하는 두 개의 쓰레드에 대한 간단한 예제를 살펴보자.

코드를 보기에 앞서 몇 가지 짚고 넘어가야 할 것이 있다. 첫 번째, 이 예제에서는 Stevens가 [SR05] 제안하듯이 쓰레드를 생성하는 루틴과 조인하는 루틴이 실패할 경우 간단하게 종료하도록 래퍼 함수를 만들었다. 이 예제와 같이 간단한 프로그램은 에러 발생 여부만 관심이 있기에, 종료 외에 다른 처리는 하지 않는다. 그렇기 때문에 `Pthread_create()` 는 단순히 `pthread_create()`를 호출하고 리턴 값이 0인지 확인한다. 만약 0이 아니면 `Pthread_create()`는 에러 메시지를 출력하고 종료한다.

두 번째, 작업자 쓰레드를 위해 두 개의 독립된 함수를 구성하는 대신 하나의 단일 코드를 사용하였다. 메시지를 출력하기 전에 각 쓰레드가 다른 문자를 출력하도록 쓰레드에게 문자열 인자를 전달하였다.

마지막으로, 그리고 가장 중요한 것은 각 작업자가 무엇을 하려는지 알 수 있다는 것이다. 각 작업자 쓰레드는 반복문 안에서 공유 변수인 `counter`에 수를 천만 번 (`1e7`) 더한다. 결과적으로 최종적으로 얻으려는 값은 `20,000,000`이다.

이제 프로그램을 컴파일하고 실행하여 결과를 확인해 보자. 때로는 실행 결과가 기대한 것과 같을 수도 있다.

```
1 prompt> gcc -o main main.c -Wall -pthread
2 prompt> ./main
3 main: begin (counter = 0)
4 A: begin
5 B: begin
6 A: done
```

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

// mythread()
// 반복문을 사용하여 단순히 1씩 더하기
// 10,000,000을 변수 counter에 더하는 방법이 아니다.
// 하지만, 문제가 무엇인지 명확하게 해준다.
void *
mythread(void *arg)
{
   printf("%s: begin\n", (char *) arg);
   int i;
   for (i = 0; i < 1e7; i++) {
      counter = counter + 1;
   }
   printf("%s: done\n", (char *) arg);
   return NULL;
}

// main()
// 두 개의 쓰레드를 실행하고 (pthread_create)
// 대기한다 (pthread_join)
int
main(int argc, char *argv[])
{
   pthread_t p1, p2;
   printf("main: begin (counter = %d)\n", counter);
   Pthread_create(&p1, NULL, mythread, "A");
   Pthread_create(&p2, NULL, mythread, "B");

   // 쓰레드가 종료할 수 있도록 대기 중인 쓰레드를 병합 한다
   Pthread_join(p1, NULL);
   Pthread_join(p2, NULL);
   printf("main: done with both (counter = %d)\n", counter);
   return 0;
}
```

〈그림 29.6〉 **데이터의 공유: Uh Oh (t1.c)**

```
B: done
main: done with both (counter = 20000000)
```

하지만 이 예제 코드를 실행하면 단일 프로세서라 하더라도 기대한 대로 결과가 출력되지는 않는다. 때로는 아래와 같은 결과를 얻는다.

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

정말 뭔가 잘못된 것인지 확인해 보기 위해 프로그램을 다시 한 번 실행해 보자. 결국에는 우리가 배웠던 것처럼 컴퓨터라면 **결정론적**인 결과를 내야하지 않겠는가?! 아니면 교수님이 거짓말을 하고 있다는 말인가?(헉)

```
prompt> ./main
main: begin (counter = 0)
```

팁 : 도구를 제대로 알고 사용하자

프로그램을 작성하고, 디버그 하고 컴퓨터 시스템의 이해를 돕는 새로운 도구들을 항상 배워야 한다. 역어셈블러라는 유용한 도구가 있다. 실행 파일을 대상으로 역어셈블러를 실행하면 해당 프로그램이 어떤 어셈블리 명령어들로 구성되었는지 알 수 있다. 예를 들어, 예제에서 사용했던 카운터를 갱신하는 저수준 코드를 이해하고 싶다면 `objdump` 라는 Linux 명령어를 실행하여 어셈블리 코드를 볼 수 있다.

```
prompt $>$ objdump -d main
```

이 명령어를 수행하면 프로그램의 모든 명령어들이 출력된다. 컴파일할 때 `-g` 명령어를 사용하였다면 프로그램의 심벌 정보를 포함하는 식별자와 함께 정리되어 나열된다. `objdump`라는 프로그램은 반드시 사용법을 배워야 하는 많은 도구들 중의 하나이다. `gdb`와 같은 디버거, `valgrind`와 `purify`와 같은 메모리 프로파일러, 그리고 컴파일러 역시 시간을 들여 사용법을 익혀야 할 도구들이다. 도구를 잘 사용할수록 더 좋은 시스템을 개발할 수 있게 된다.

```
3 A: begin
4 B: begin
5 A: done
6 B: done
7 main: done with both (counter = 19221041)
```

각 실행이 잘못되었을 뿐만 아니라 각 실행의 결과 역시 *다르다*! 아주 커다란 의문이 남는다 : 왜 이런 일이 일어나는 것일까?

29.3 제어 없는 스케줄링

왜 이런 현상이 발생하는지 이해하려면 `counter` 갱신을 위해서 컴파일러가 생성한 코드의 실행 순서를 이해해야 한다. 카운터에 단순히 (1) 이라는 숫자를 더하려고 한다. x86에서 `counter`를 증가하는 코드의 순서는 다음과 같다.

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

이 예제에서 사용하는 `counter` 변수의 위치의 주소는 `0x8049a1c`라고 가정한다. 이 세 줄의 명령문에서는 먼저 x86의 mov 명령어가 명시한 메모리 주소의 값을 읽어들인 후 `eax` 레지스터에 넣는다. 그리고 1 (`0x1`)을 `eax` 레지스터의 값에 더하는 연산을 한 후에 마지막으로 `eax`에 저장되어 있는 값을 메모리의 원래의 주소에 다시 저장한다.

두 개의 쓰레드 중에 쓰레드 1이 `counter`를 증가시키는 코드 영역에 진입하여 `counter`의 값을 1 증가시키려는 상황을 가정해 보자. `counter`에 있는 값이 50이었다고 하면 50을 `eax`레지스터에 넣는다. 쓰레드 1에 있어서 `eax=50`이 된다. 그 후에 레지스터의 값에 1을 더하여 `eax=51`이 된다. 이제 상황이 안 좋아진다. 타이머 인터럽트가 발생하여 운영체제가 실행 중인 쓰레드의 `PC` 값과 `eax`를 포함하는 레지스터들

등의 현재 상태를 쓰레드의 TCB에 저장한다.

그리고는 상황이 더 악화된다. 쓰레드 2가 선택되고 `counter` 값을 증가시키는 똑같은 코드 영역에 진입한다. 첫 번째의 명령어를 실행하여 `counter` 값을 얻어 온 후 `eax`에 넣는다. 쓰레드는 개별적으로 쓰레드 전용 레지스터를 가지고 있다. 사용 중이던 레지스터들을 저장하고 복구하는 문맥 교환 코드에 의해 이 레지스터들은 가상화된다. `counter` 값은 아직 50을 나타내고 있어서 쓰레드 2의 `eax`의 값은 50이다. 쓰레드 2가 그 다음의 두 문장을 실행한다면 `eax` 값을 1 증가시키고 (`eax=51`) 난 후에 `eax` 값을 `counter`(주소 `0x8049a1c`)에 저장한다. 전역 변수인 `counter`는 이제 51이라는 값을 가진다.

최종적으로 또 한 번의 문맥 교환이 발생하면 쓰레드 1이 리턴하여 실행된다. 이전 상황을 기억해 보면 쓰레드 1은 `mov`와 `add` 동작을 실행하였고 이제 마지막의 `mov` 명령어를 수행하려는 중이다. 그리고 `eax=51` 이었다. 그렇기 때문에 `mov` 명령어가 실행되면 메모리에 레지스터의 값을 저장하여 `counter`의 값은 다시 51이 된다.

무슨 일이 발생한 것인지 간단히 이야기 하면 `counter`의 값을 증가시키는 코드가 두 번 수행이 되었지만 50에서 시작한 `counter`의 값은 1 증가한 51이다. "정확하게" 동작하도록 작성된 프로그램이라고 한다면 `counter`의 값은 52가 되어야 한다.

이 문제를 좀 더 잘 이해하기 위해서 실행의 흐름을 구체적으로 살펴보자. 이 예제에서는 다음에 나타난 것과 같이 코드가 메모리 주소 100에 저장되었다고 가정해 보자 (RISC 유사 명령어 집합에 익숙하다면 참고해야 할 것이 있는데, x86은 명령어들의 길이는 가변적이라는 것과 `mov` 명령은 5바이트의 메모리를 사용하고 add는 3바이트를 사용한다).

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

이러한 가정했을 때 일어나는 동작을 그림 29.7에 나타내었다. `counter`의 값은 50에서 시작한다고 가정하고 예제의 내용을 상기하며 흐름을 따라가 보자.

예시에서처럼 명령어의 실행 순서에 따라 결과가 달라지는 상황을 **경쟁 조건**(**race condition**)이라고 부른다. 문맥 교환이 때에 맞지 않게 실행되는 운이 없는 경우 잘못된 결과를 얻게 된다. 사실, 경쟁 조건에 처한 경우 실행할 때마다 다른 결과를 얻는다. 컴퓨터의 작동에서 일반적으로 발생하는 **결정적** 결과와 달리 결과가 어떠할지 알지 못하거나 실행할 때마다 결과가 다른 경우를 **비결정적**(**indeterminate**)인 결과라고 부른다.

멀티 쓰레드가 같은 코드를 실행할 때 경쟁 조건이 발생하기 때문에 이러한 코드 부분을 **임계 영역**(**critical section**)이라고 부른다. 공유 변수(또는 더 일반적으로는 공유 자원)를 접근하고 하나 이상의 쓰레드에서 동시에 실행되면 안 되는 코드를 임계 영역이라 부른다.

이러한 코드에서 필요한 것은 **상호 배제**(**mutual exclusion**)이다. 이 속성은 하나의 쓰레드가 임계 영역 내의 코드를 실행 중일 때는 다른 쓰레드가 실행할 수 없도록 보장해 준다.

			(명령어 실행 후)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	임계 영역 전		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
T1 상태 저장					
T2 상태 복원			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1	113	51	**51**
interrupt					
T2 상태 저장					
T1 상태 복원			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

〈그림 29.7〉 **문제 : 가까이서 친밀하게 보기**

참고로 거의 모든 용어들은 Edsger Dijkstra에 의해 처음 만들어졌다. Dijkstra는 이 분야를 개척한 공로와 그 외의 다른 업적을 인정받아 Turing Award를 수상하였다. 언급한 문제에 대한 명쾌한 설명을 알고 싶다면 "Cooperating Sequential Processes" [Dij68]라는 1968년의 논문을 읽어 보기 바란다. Dijkstra는 이번 장에서 계속 다루어진다.

29.4 원자성에 대한 바람

임계 영역 문제에 대한 해결 방법 중 하나로 강력한 명령어 한 개로 의도한 동작을 수행하여, 인터럽트 발생 가능성을 원천적으로 차단하는 것이다. 예를 들면 다음과 같은 강력한 명령어가 있다고 해 보자.

```
memory-add 0x8049a1c, $0x1
```

이 명령어는 메모리 상의 위치에 어떤 값을 더하는 명령어다. 하드웨어는 이 명령어가 원자적으로 실행되는 것을 보장한다고 하자. 이 명령어가 실행되면 원하는 것처럼 값이 갱신될 것이다. 하드웨어가 원자성을 보장해주기 때문에 명령어 수행 도중에 인터럽트가 발생하지 않는다. 인터럽트가 발생하더라도, 명령어가 실행이 안되었거나 실행이 종료된 후라는 것을 의미하고 그 중간 상태는 있을 수 없다. 멋진 하드웨어 아닌가?

문맥 상으로 원자적이라는 말은 "하나의 단위"를 뜻하며 때로는 "전부 아니면 전무"로 이해될 수도 있다. 다음의 세 개의 명령어가 원자적으로 실행되기를 원한다.

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

여담 : 주요 병행성 관련 용어
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

사용된 네 개의 용어는 병행 코드에 중심이 되는 용어들이기 때문에 명시적으로 정의하고 가도록 한다. 더 자세한 정보를 위해서는 Dijkstra의 초기 연구 [Dij65; Dij68]들을 살펴보기 바란다.

- **임계 영역**(**critical section**)은 보통 변수나 자료 구조와 같은 공유 자원을 접근하는 코드의 일부분을 말한다.
- **경쟁 조건**(**race condition**)은 멀티 쓰레드가 거의 동시에 임계 영역을 실행하려고 할 때 발생하며 공유 자료 구조를 모두가 갱신하려고 시도한다면 깜짝 놀랄 의도하지 않은 결과를 만든다.
- **비결정적**(**indeterminate**) 프로그램은 하나 또는 그 이상의 경쟁 조건을 포함하여 그 실행 결과가 각 쓰레드가 실행된 시점에 의존하기 때문에, 프로그램의 결과가 실행할 때마다 다르다. 결과는 컴퓨터 시스템에서 일반적으로 기대하는 바와 달리 **결정적**이지 않다.
- 이와 같은 문제들을 회피하려면 쓰레드는 **상호 배제**(**mutual exclusion**)라는 기법의 일종을 사용하여서 하나의 쓰레드만이 임계 영역에 진입할 수 있도록 보장한다. 그 결과로 경쟁을 피할 수 있고 프로그램 실행 결과를 결정론적으로 얻을 수 있게 된다.

언급했듯이, 위의 명령어들을 하나의 명령어로 대신할 수 있다면 그 명령어를 사용하면 그만이다. 하지만 일반적인 상황에서는 그러한 명령어가 없다고 봐야 한다. 병행성을 가지는 B-tree를 만드는 중이고, 값을 갱신한다고 했을 때, 원자적으로 B-tree를 갱신하는 어셈블리 명령어를 원할까? 글쎄, 아마도 제정신이라면 아니라고 본다.

하드웨어적으로는 **동기화 함수**(**synchronization primitives**) 구현에 필요한 기본적인 명령어[2] 몇 개만 필요하다. 결과적으로 병행 실행이라는 어려운 상황에서 하드웨어 동기화 명령어와 운영체제의 지원을 통해 한 번에 하나의 쓰레드만 임계 영역에서 실행하도록 구성된, "제대로 잘 작동하는" 멀티 쓰레드 프로그램을 작성할 수 있다. 멋지지 않은가?

이것이 이번 장에서 다루게 될 문제이다. 어려운 문제들이라서 골치가 (약간은) 아프게 될 것이다. 골치가 안 아프면 제대로 이해한 것이 아니다. 계속 공부해서 머리가 아프게 되면 그때서야 제대로 된 방향으로 가고 있구나 하면 된다. 머리 아프게 만들려는 것이 아니니, 그때가 되면 잠시 쉬기 바란다.

2) 역자 주 : 어셈블리 명령어

핵심 질문: 동기화를 지원하는 방법
유용한 동기화 함수를 만들기 위해 어떤 하드웨어 지원이 필요한가? 운영체제는 어떤 지원을 해야 하는가? 어떻게 하면 이런 함수를 정확하고 효율적으로 만들 수 있을까? 프로그램들이 이 함수들을 활용하여 의도한 결과를 얻으려면 어떻게 해야 할까?

29.5 또 다른 문제: 상대 기다리기

이제까지 병행성 문제를 공유 변수 접근에 관련된 쓰레드 간의 상호 작용 문제로 정의하였다. 실제로는 하나의 쓰레드가, 다른 쓰레드가 어떤 동작을 끝낼 때까지 대기해야 하는 상황도 빈번하게 발생한다. 프로세스가 디스크 I/O를 요청하고 응답이 올 때까지 잠든 경우가 좋은 예이다. I/O 완료 후 잠들었던 프로세스가 깨어나 이후 작업을 진행한다.

이후에서는 원자성 지원을 위한 동기화 함수 제작에 대한 내용과 멀티 쓰레드 프로그램에서 흔한 잠자기/깨우기 동작에 대한 지원 기법에 대해서 다룬다. 지금 이해가 가지 않더라도 괜찮다. **컨디션 변수**(**condition variable**)에 대한 장을 읽을 때 즈음에는 이해가 될 것이다. 그때까지도 이해가 안 된다면, 그 장이 이해가 될 때까지 반복하여 읽어서 이해해야 한다. 讀書百遍而義自見[3]이다.

29.6 정리: 왜 운영체제에서?

정리에 앞서 왜 이런 것들을 운영체제에서 다뤄야 하는지 의문이 들 수도 있다. 한 단어로 대답하자면 "역사"이다. 운영체제는 최초의 병행 프로그램이었고 운영체제 내에서 사용을 목적으로 다양한 기법들이 개발되었다. 나중에는 멀티 쓰레드 프로그램이 등장하면서 응용 프로그래머들도 이러한 문제를 고민하게 되었다.

예를 들어 두 개의 프로세스가 실행 중인 경우를 생각해 보자. 임의의 파일에 둘 다 `write()`를 수행하여 데이터를 파일에 덧붙이려고 한다. 데이터를 파일의 끝 부분에 붙이게 되면 파일의 길이가 증가한다. 두 프로세스는 새로운 블럭을 할당받고, 파일 `inode`에 블럭의 위치와 변경된 크기를 기록한다(파일에 대해서는 책의 제3편에서 배우게 될 것이다). 인터럽트가 언제든지 발생할 수 있기 때문에 이러한 공유 자료 구조(예: 할당을 위한 비트맵 또는 파일의 `inode`)를 갱신하는 코드는 임계 영역이 된다. 인터럽트가 처음 소개되었을 때부터 운영체제 설계자들은 운영체제가 어떻게 내부 구조를 갱신할 것인가에 대해 고민해 왔다. 시도 때도 없이 발생하는 인터럽트가 앞서 언급한 모든 문제들의 원인이다. 페이지 테이블, 프로세스 리스트, 파일 시스템 구조 그리고 대부분의 커널 자료 구조들이 올바르게 동작하기 위해서는 적절한 동기화 함수들을 사용하여 조심스럽게 다루어져야 한다.

3) 독서백편이의자견 - 삼국지 위지(魏志)편. 조조가 세운 위나라에 황문시랑(黃門侍廊)이란 벼슬에 올라 헌제(獻帝)의 글공부 상대였던 동우(董遇)는 글을 배우겠다고 오는 사람에게 "내게서 배우기보다는 집에서 혼자 읽고 또 읽어 보면 뜻을 알게 될 것이다."라 하고, 그 사람을 보냈다고 한다.

여담: 원자적 연산(atomic operation)을 사용하자

원자적 연산들은 컴퓨터 시스템을 이루는 가장 강력한 기술 중에 하나로서 컴퓨터 구조와 우리가 지금 다루고 있는 병행 코드, 곧 다루게 될 파일 시스템들과 데이터베이스 관리 시스템들 그리고 분산 시스템들의 근간을 이룬다 [Lyn+93].

연속된 동작들을 **원자적**으로 만든다는 개념은 간단하게 "전부 아니면 전무"라고 표현할 수 있다. 수행하려는 모든 동작이 모두 다 처리된 것처럼 보이거나 실행되다가 만 중간 상태가 없도록 아무 것도 실행되지 않은 것처럼 보여야 한다. 때로는 여러 동작을 묶어 하나의 원자적 동작이 되도록 만든 것을 트랜젝션이라고 부른다. 이 개념은 데이터베이스와 **트랜젝션** 처리 [GR92] 분야에서 심도 있게 다루고 있다.

병행성에 대한 이번 주제에서 동기화 함수를 사용하여 순차적인 명령어들을 원자적 실행 단위로 만들 것이다. 원자성이라는 개념은 훨씬 더 포괄적이다. 예를 들어 파일 시스템은 저널링(journaling)이나 쓰기-시-복사(Copy-On-Write)와 같은 기법을 사용하여 디스크 상태를 원자적으로 전이시킨다. 시스템 오류가 발생하더라도 올바르게 동작하기 위해서 원자적으로 상태를 전이시키는 것은 필수적이다. 이해가 안되더라도 앞으로 이해하게 될 테니 걱정하지 말라.

참고 문헌

[Dij68] **"Cooperating sequential processes"**
Edsger W. Dijkstra
URL: `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`
Dijkstra는 그가 마지막으로 일을 했던 University of Texas에 있는 이 사이트에 굉장히 많은 수의 논문과 기록물, 그리고 생각들을 (후대를 위해) 남겨 놓았다. 하지만, 유명한 "cooperating sequential processes" 논문을 포함한 대부분의 기초 연구들은 Dijkstra가 Technische Hochshule of Eindhoven(THE)에 있을 때 이루어졌다. 이 논문에서는 멀티 쓰레드 프로그램을 작성하는데 필요한 모든 개념들을 정리하였다. Dijkstra는 THE 운영체제에 관해 연구할 때 멀티 쓰레드 프로그램에 관한 개념 대부분을 발견하였다. 이 운영체제는 Dijkstra가 재직하던 학교의 이름을 땄다 (관사처럼 "the"로 읽는 것이 아니라 개별적으로 "T", "H", "E" 라고 읽음)

[Dij65] **"Solution of a problem in concurrent programming control"**
E.W. Dijkstra
Communications of the ACM, 8(9):569, September 1965
Dijkstra가 상호 배제의 문제와 해결책을 정리한 첫 번째 논문이다. 하지만, 제시된 해결법이 많이 사용되고 있지는 않다. 앞으로 보게 되겠지만, 개선된 하드웨어와 운영체제의 지원이 필요하다.

[GR92] **"Transaction Processing: Concepts and Techniques"**
Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
트랜젝션 처리 분야의 전설적인 인물인 Jim Gray가 쓴 이 분야의 성서와 같은 책이다. 이러한 이유로 이 책은 Jim Gray의 브레인 덤프(brain dump)라고도 알려져 있다. Gray는 이 책에 자신이 알고 있는 데이터베이스 관리 시스템의 동작에 관한 모든 것을 기록하였다. 안타깝게도 Gray는 몇 해 전 비극적으로 사망하였으며 운 좋게도 Gray와 대학원 시절에 교류할 수 있었던 이 책의 공동저자를 포함하여 우리는 친구이자 위대한 멘토를 잃었다.

[Lyn+93] **"Atomic Transactions"**
Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete

Morgan Kaufmann, August 1993
분산 시스템을 위한 원자적 트랜젝션에 대한 이론과 실제에 대한 좋은 텍스트이다. 어떤 이에게는 수식이 너무 많다고 할 수 있겠지만 상당히 좋은 내용들을 다루고 있다.

[SR05] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
여러 차례 언급했던 것과 같이 이 책은 꼭 구입해서 자기 전에 몇 쪽씩 읽기를 권한다. 더 빨리 잠에 빠질 수 있을 뿐 아니라 깊이 있는 UNIX 프로그래머가 되는 법을 조금이라도 더 배울 수 있게 해준다.

숙제

x86.py은 쓰레드 실행 순서가 어떻게 경쟁 조건을 유발하거나 회피하는지 볼 수 있는 프로그램이다. 이 프로그램의 동작 방법과 기본 입력 값에 대한 설명을 원한다면 README 파일을 읽어라. 그런 후 다음 질문들에 대해 답해보라.

문제

1. 우선 "loop.s"라는 간단한 프로그램을 먼저 살펴보자. cat loop.s 명령어로 프로그램을 먼저 읽어보고 이해할 수 있는지 보자. 그리고 다음의 인자 값을 사용하여 실행해 보자.

   ```
   ./x86.py -p loop.s -t 1 -i 100 -R dx
   ```

   위 명령어의 옵션은 한 개의 쓰레드, 100개의 명령어를 실행할 때마다 인터럽트 발생, %dx의 값을 추적하라는 옵션이다. 실행 중에 %dx의 값이 무엇일지 알 수 있겠는가? 그 값을 예상했다면, -c 플래그와 함께 다시 실행하여 답을 확인해 보자. 오른편의 명령어가 실행되면 왼편에 레지스터의 값(또는 메모리 값)이 나타난다.

2. 이제 다음과 같은 플래그를 사용하여 똑같은 코드를 실행해 보자.

   ```
   ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
   ```

   이 명령어는 두 개의 쓰레드를 지정하고 각각 %dx 레지스터의 값을 3으로 만든다. 어떤 값이 %dx에 저장되겠는가? -c 플래그와 함께 실행하여 답을 확인해 보자. 쓰레드가 여러 개라는 사실이 당신의 추측에 영향을 주는가? 이 코드에 경쟁 조건이 있는가?

3. 다음의 조건으로 실행해 보자.

   ```
   ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
   ```

   위의 명령어는 인터럽트가 짧은 간격으로 아무 때나 발생하도록 지정한다. 다른 순서의 쓰레드 실행을 보고 싶다면 -s 옵션으로 새로운 시드를 사용하라. 인터럽트 실행 빈도를 변경하는 것이 프로그램의 결과에 영향을 주는가?

4. 다음으로 looping-race-nolock.s 라는 다른 프로그램을 살펴보자. 이 프로그램은 2000번지의 메모리 주소에 위치한 공유 변수에 접근한다. 간단하게 하기 위해서 이 변수를 x라고 하겠다. 다음과 같이 단일 쓰레드로 실행하고, 어떤 일을 하는지 확실하게 이해하자.

   ```
   ./x86.py -p looping-race-nolock.s -t 1 -M 2000
   ```

   전체 실행 시간 동안 x 의 값은 무엇인가(즉, 메모리 2000번지의 값)? -c를 사용하여 답을 확인해 보자.

5. 이번에는 쓰레드의 개수를 늘리고 반복 횟수도 늘려서 실행해 보자.

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

각 쓰레드의 코드가 왜 세 번씩 반복하는지 이해되는가? **x**의 최종 값은 무엇이 되겠는가?

6. 랜덤 간격으로 인터럽트가 발생하도록 하여 실행해 보자.

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

실행한 후에 랜덤 함수의 시드를 `-s 1` 로 해보고 `-s 2` 등으로 변경해 보자. 쓰레드 실행 순서를 보고 **x**의 최종 값이 무엇인지 알 수 있겠는가? 인터럽트의 정확한 발생 위치가 중요한가? 인터럽트가 발생해도 안전한 위치는 어디인가? 인터럽트가 문제를 일으키는 위치는 어디인가? 다시 말해 정확하게 어디가 임계 영역인가?

7. 이번에는 인터럽트가 일정 간격으로 발생하도록 지정하고 프로그램의 동작을 살펴보자.

```
./x86.py -p looping-race nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

공유 변수 x의 최종 값을 예측할 수 있는지 한 번 해 보라. `-i 2` 나 `-i 3` 와 같이 변경했을 때는 가능하겠는가? 인터럽트의 간격을 얼마로 했을 때 "올바른" 최종 정답을 얻을 수 있는가?

8. 반복 횟수를 늘려서 (예, `-a bx=100`) 위의 명령어를 실행해 보자. `-i` 플래그로 설정되는 인터럽트의 간격을 얼마로 정할 때 "올바른" 결과를 나타내는가? 간격이 얼마일 때 예기치 못한 결과가 나오는가?

9. 이 숙제의 마지막으로 `wait-for-me.s` 프로그램을 살펴보려고 한다. 다음과 같이 코드를 실행해 보자.

```
./x86.py -p wait-for-me.s -a ax=1, ax=0 -R ax -M 2000
```

쓰레드 0의 %**ax** 레지스터는 1로 설정하고 쓰레드 1의 레지스터는 0으로 설정한 후에 프로그램이 실행하는 동안 %**ax**의 값과 메모리 2000번지에 있는 값을 주시하라. 코드는 어떤 행동을 보여야 하는가? 쓰레드들은 2000번지의 값을 어떻게 사용하는가? 최종 값은 무엇이 되겠는가?

10. 입력 인자를 다음과 같이 바꿔보자

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
```

쓰레드들은 어떤 행동을 보이는가? 쓰레드 0은 무엇을 하는가? 인터럽트 간격을 변경하면 (예, `-i 1000` 또는 랜덤 간격으로) 실행 추적 결과는 어떻게 달라지는가? 프로그램이 CPU를 효율적으로 사용하고 있는가?