상호배제 알고리즘에 관한 연구

최성민*, 이형봉*
*강릉원주대학교 컴퓨터공학과
e-mail:{csm5588, hblee}@gwnu.ac.kr

A Study on Mutual Exclusion Algorithms

Seong-Min Choi*, Hyung-Bong Lee*
*Dept of Computer Science & Engineering, Gangneung-Wonju National University

요 약

운영체제 수업 내용 중 가장 흥미로우면서도 이해하기가 어려운 부분이 상호배제 알고리즘이다. 이 논문에서는 상호배제 알고리즘으로 널리 알려진 Dekker's 알고리즘과 Peterson's 알고리즘을 C 언어 환경에서 실험하는 과정에서 겪은 시행착오를 공유함으로써 보다 효율적인 학습에 도움을 주고자 한다. 또한, Dekker's 알고리즘의 개선으로 이루어진 Peterson's 알고리즘은 성능 관점에서는 오히려 크게 저조하게 나타났는데 그 원인을 분석한다.

1. 서론

프로세스 관리, 프로세서 관리, 프로세스 동기화, 교착상태 관리, 메모리 관리, 파일시스템 관리, 입출력 관리, 디스크 스케줄링 등의 내용으로 이루어진 운영체제 교과 내용 중, 프로세스 동기화 부분은 흥미롭기도 하면서 이해가 어렵지만 소프트웨어 개발에서의 활용 가치가 크다. 프로세스 동기화의 주요 내용은 다수의 프로세스 혹은 스레드들이 공유변수 등 공유자원에 접근할 때 발생하는 경쟁상황을 관리하여 올바른 처리 결과를 보장하기 위한 기법을 다룬다[1]. 이 논문에서는 공유자원에 접근하는 임계영역을 보호하는 소프트웨어적 상호배제 알고리즘으로 널리 알려진 Dekker's 알고리즘과 Peterson's 알고리즘을 C 언어로 실험하는 과정에서 발생했던 오류를 정리하여 시행착오를 줄일 수 있도록 한다. 또한, 이들 두 알고리즘은 기능적으로는 동일하지만, 실험 과정에서 Peterson's 알고리즘 성능상이 크게 저조하다는 점이 발견되었는데 그 원인을 운영체제 과목의 프로세서 관리 관점에서 분석한다.

2. Dekker's 알고리즘과 Peterson's 알고리즘 실험 환경 및 시행착오

상호배제가 필요한 가상 환경으로 이 논문에서는 그림 1의 프로그램을 사용한다. 이 프로그램은 유닉스 혹은 리눅스 운영체제에서 초기 값 0인 공유 변수 Count에 각각 50,000,000번 1을 더하고 빼는 두 개의 POSIX 스레드로 구성되어 있고, Count의 최종 값은 0이어야 한다.

```
#include <stdio.h>
#include <pthread.h>
int          Count = 0;
변수정의
void *add(void *ap) /* P0 */
{
    int      n, v, i;
    n = (int)ap;
    for (i = 0; i < n; i++) {
             v = do_something();
             잠금코드
             Count += v; /* 임계영역 */
             해제코드
    }
    pthread_exit((void *)NULL);
}
```

```
int do_something()
{
    return (1);
}
void *sub(void *ap) /* P1 */
{
    int      n, v, i;
    n = (int)ap;
    for (i = 0; i < n; i++) {
             v = do_something();
             잠금코드
             Count -= v; /* 임계영역 */
             해제코드
    }
    pthread_exit((void *)NULL);
}
```

```
main(int ac, char *av[])
{
    pthread_t              thread1, thread2;

    pthread_create(&thread1, NULL, add,
                        (void *)50000000);
    pthread_create(&thread2, NULL, sub,
                        (void *)50000000);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Count = %d₩n", Count);
}
```

(그림 1) Dekker's 및 Peterson's 상호배제 알고리즘 실험을 위한 C 언어 코드

□ 시행착오 1

변수정의	P0 잠금코드	P1 잠금코드

int Flag0 = 0;
int Flag1 = 0;
int Turn = 0;
	Flag0 = 1;
	while(Flag1 != 0) {
	if (Turn == 1) {
	Flag0 = 0;
	while(Turn == 1);
	Flag0 = 1;
	}
	}
	P0 해제코드
	Turn = 1;
	Flag0 = 0;
		Flag1 = 1;
		while(Flag0 != 0) {
		if (Turn == 0) {
		Flag1 = 0;
		while(Turn == 0);
		Flag1 = 1;
		}
		}
		P1 해제코드
		Turn = 0;
		Flag1 = 0;

(그림 2) Dekker's 알고리즘의 C 언어 코드

그림 1에 적용할 Dekker's 알고리즘을 단순히 참고문헌에 나타나 있는 대로 그림 2와 같이 구현하여 실험하면, Count 변수의 최종 값이 올바르게 0이 출력되지 않는다. 그 이유는 두 스레드 P0와 P1에서 공유되는 세 개의 변수 Flag0, Flag1, Turn의 현재 값이 컴파일러의 최적화 과정으로 인해 메모리에서 참조되지 않고 각 스레드의 문맥으로 CPU 내부에서 유지되고 참조되기 때문이다. 이를 해결하기 위해서 이들 변수들을 반드시 메모리에서 참조하도록 volatile int 타입으로 선언해준다. 이 때 간과하기 쉬운 사항으로 Count도 공유변수이기 때문에 이 역시 volatile 타입으로 수정해주어야 한다.

□ 시행착오 2

변수정의	P0 잠금코드	P1 잠금코드

volatile int Flag0 = 0;
volatile int Flag1 = 0;
volatile int Turn = 0;
	Flag0 = 1; Turn = 1;
	while(Flag1 && Turn == 1)
	;
	P0 해제코드
	Flag0 = 0;
		Flag1 = 1; Turn = 0;
		while(Flag0 && Turn == 0)
		;
		P1 해제코드
		Flag1 = 0;

(그림 3) Peterson's 알고리즘의 C 언어 코드

그림 1에 적용할 Peterson's 알고리즘을 그림 3과 같이 평범하게 구현하여 실험하면 시간이 너무 많이 소요되어 실험이 거의 불가능하다. 그 이유는 어느 한 쪽 스레드에게 CPU가 할당되었을 때 마침 while 코드에서의 바쁜 대기가 필요하면 주어진 타임 슬롯을 모두 바쁜 대기로 소진한 다음에야 다른 스레드로의 스케줄링이 이루어지기 때문이다. 이를 완화시키기 위해서 바쁜 대기 while 구문에 sched_yield() 삽입하여 CPU를 양보하도록 한다.

3. Dekker's 알고리즘과 Peterson's 알고리즘의 성능 비교

시행착오 1, 2를 반영하여 Dekker's와 Peterson's 알고리즘을 그림 1에 각각 적용하여 HP DS20 유닉스 시스템에서 실행시간을 측정하면 두 알고리즘 사이에 현격한 차이가 발견되는데, 그림 3에 반복 횟수에 따른 실행시간 추이를 보였다. 이 표에서 Peterson's 상호배제 알고리즘의 시간 소모가 많은 원인은 CPU 스케줄링과 관계있는 것으로, 아래와 같이 분석된다.

▫ 프로세스(스레드)에게 CPU가 할당되었을 때, 할당된 시간 동안 최대한 많은 덧셈 혹은 뺄셈의 반복이 이루어질 수 있어야 한다. 즉, 바쁜 대기 상태에서 시간 소모가 적어야 한다.

▫ Dekker's 알고리즘에서는 두 프로세스가 거의 동시에 잠금 코드에 접근했을 때, 현재의 Turn에 따른 후순위 프로세스가 Flag를 내려주므로 다른 프로세가 Turn에 관계없이 주어진 타임 슬라이스 동안 반복적 재진입이 가능하다.

<표 1> Dekker's 알고리즘과 Peterson's 알고리즘의 상호배제 시간 측정 (단위: sec)

알고리즘 \ 시도횟수	10,000,000	20,000,000	30,000,000	40,000,000	50,000,000
Dekker's	0.48	0.97	1.48	2.03	2.64
Peterson's	42.32	98.65	144.26	190.61	257.31

▫ Peterson's 알고리즘에서는 Dekker's 알고리즘과 다르게 후순위 프로세스가 바쁜 대기만 할 뿐 Flag는 그대로 유지하므로 결국은 Turn에 따라 순위가 결정된다. 그런데, Turn은 진입을 시도할 때마다 수정되므로 결국은 두 프로세스가 번갈아 한 번씩 진입에 성공하고, 그 때마다 문맥교환을 실시하는 운영체제의 부담으로 인해 실행시간이 길어진다.

4. 결론

의사코드로 기술된 Dekker's 알고리즘과 Peterson's 알고리즘을 활용할 때는 구현 언어와 운영체제 환경에 따른 세밀한 보완을 필요로 한다는 점을 발견하였다. 특히, 라운드로빈 형태의 시분할 운영체제에서 Peterson's 알고리즘은 근본적인 수정이 필요하다고 판단되어 연구를 더 진행할 예정이다.

참고문헌

[1] 이형봉, "운영체제 이론과 실제", pp. 105-휴먼사이언스

[2] Wikipedia, https://en.wikipedia.org/wiki/Dekker%27s_algorithm

% 이 논문은 2016년 강릉원주대학교 서울어코드활성화사업단의 지원을 받아 작성되었음(교신저자: 이형봉)