시스템 프로그래밍 실습 Report

실습 제목: Proxy #3-1

Synchronize Shared Resource

실습일자: 2022년 5월 16일 (월)

제출일자: 2022년 6월 01일 (수)

학 과: 컴퓨터정보공학부

담당교수: 최상호 교수님

실습분반: 목요일 7,8교시

학 번: 2018202065

성 명: 박 철 준

- Introduction

1. 제목

Proxy 3-1

Synchronize Shared Resource

2. 목표

- 과거 2-4 과제 부분에 logfile에 관한 동시접근 제어 추가한다.
- semkey 값은 port number와 같게 하여야 한다.
- Semaphore를 사용하여 한 번에 하나의 프로세스만 log file에 기록하도록 수정하여야 한다.
- 동시에 여러 프로세스가 접근했을 때를 시뮬레이션하여 서버의 터미널에 상태 출력하여야 한다.

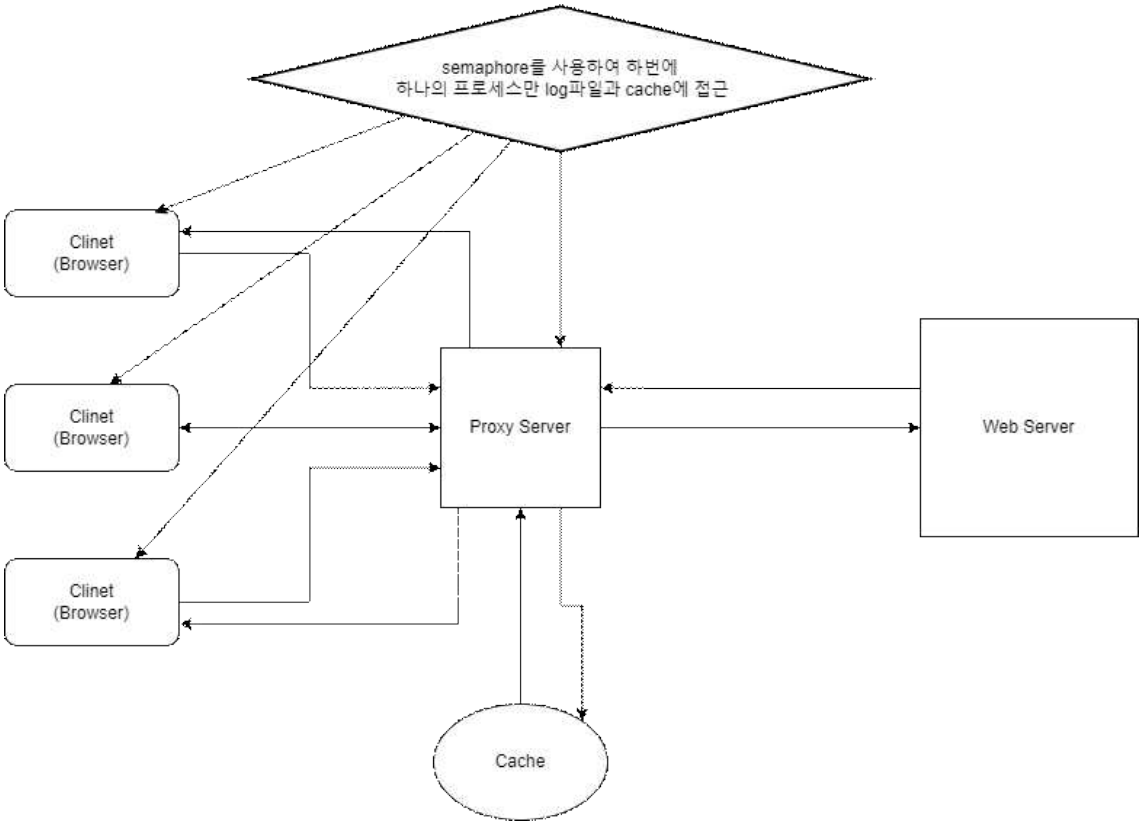
3. 실험 방법

- 시뮬레이션 시나리오를 작성해서 실험하여야 한다.
- Critical section 내에 sleep() 등을 사용하면 동시 접근하는 경우를 볼 수 있다.
- 터미널 출력 화면에 출력하여야 하며 터미널 양식은 아래와 같다.
  - *PID# getpid() is waiting for the semaphore.
  - *PID# getpid() is in the critical zone.
  - *PID# getpid() exited the critical zone.

● 실습 구현

-Flow Chart

세마포어를 이용하여 하나의 프로세스만이 Critical section을 지정하여 공유자원인 log파일과 cache파일에 접근 할 수 있는 흐름도를 가졌다.



-Pseudo code

이번 과제의 Pesudo code는 이전 과제에서 설명한 것은 제외하고 구현한 p 함수, v 함수 그리고 main함수의 추가 수정부분을 설명하고자 한다. main 함수에 있어 이전 과제와 달라진 부분은 세마포어를 사용하여 프로세스 공유 리소스를 통제하는 과정이 추가 되었다. 이는 아래와 같다.

main 함수

코드가 긴 관계로 추가한 부분만 설명하고자 한다.

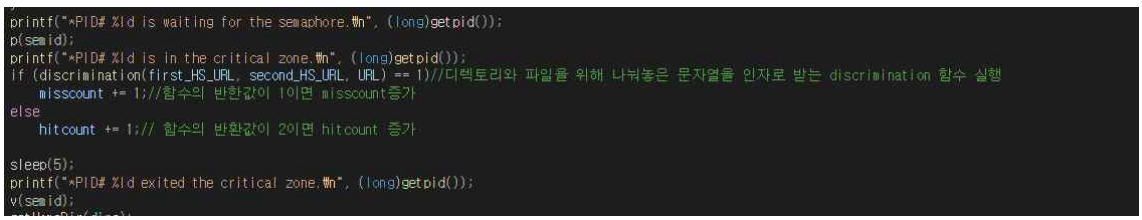
```
int semid;
//핸들러에서 세마포어의 삭제를 위해 세마포어의 아이디 저장은 전역변수로 선언한다.
union semum
//핸들러에서 세마포어의 삭제를 위해 semum은 전역변수로 선언한다.
{
        int val;
        struct semid_ds* buf;
        unsigned short int* array;
```

```
}arg;
int main()
{
        //main 함수로 리눅스에서 실행 되며 위에 정의한 함수들을
        // 이용하여 본 과제의 목적인 Construction Proxy Connection을 구현한다.
        // proxy 서버통신에 있어 proxy 서버 역할을 한다.
        if ((semid = semget((key_t)80, 1, IPC_CREAT | 0666)) == -1)
        //세마포어 아이디를 설정
        {
                //에러이면
                perror("semget failed");
                exit(1);
        }
        arg.val = 1;
        if ((semctl(semid, 0, SETVAL, arg)) == -1)
        //세마포어 값을 초기화 한다.
        {
                //에러이면
                perror("semctl failed");
                exit(1);
        }
        ...(생략 2-4와 동일)
}

p() 함수
void p(int semid)
{
        struct sembuf pbuf; //세마포어 구조체 선언
        pbuf.sem_num = 0; // 세마포어 구조체에서 세마포어 순서 설정
        pbuf.sem_op = -1; //세마포어 값을 –1하는 연산 설정
        pbuf.sem_flg = SEM_UNDO; // 프로그램이 종료되어도 자동으로 세마포어 삭제
        if ((semop(semid, &pbuf, 1)) == -1)// 세마포어 값을 음수로 만들어 세마포어를
        사용중으로 바꿈 크리티컬 섹션 이용 불가능
        {
                //오류이면
                perror("p : semop failed");
                exit(1);

        }

}
```

```
void v(int semid)
{
        struct sembuf vbuf;//세마포어 구조체 선언
        vbuf.sem_num = 0;// 세마포어 구조체에서 세마포어 순서 설정
        vbuf.sem_op = 1; // 세마포어 값을 +1하는 연산 설정
        vbuf.sem_flg = SEM_UNDO;//프로그램이 종료되어도 자동으로 세마포어 삭제
        if ((semop(semid, &vbuf, 1)) == -1)
        //세마포어 값을 양수로 만들어 세마포어를 사용가능으로 바꿈 크리티컬
        섹션 사용 가능
        {
                //오류이면
                perror("v : semop failed");
                exit(1);
        }
}
```

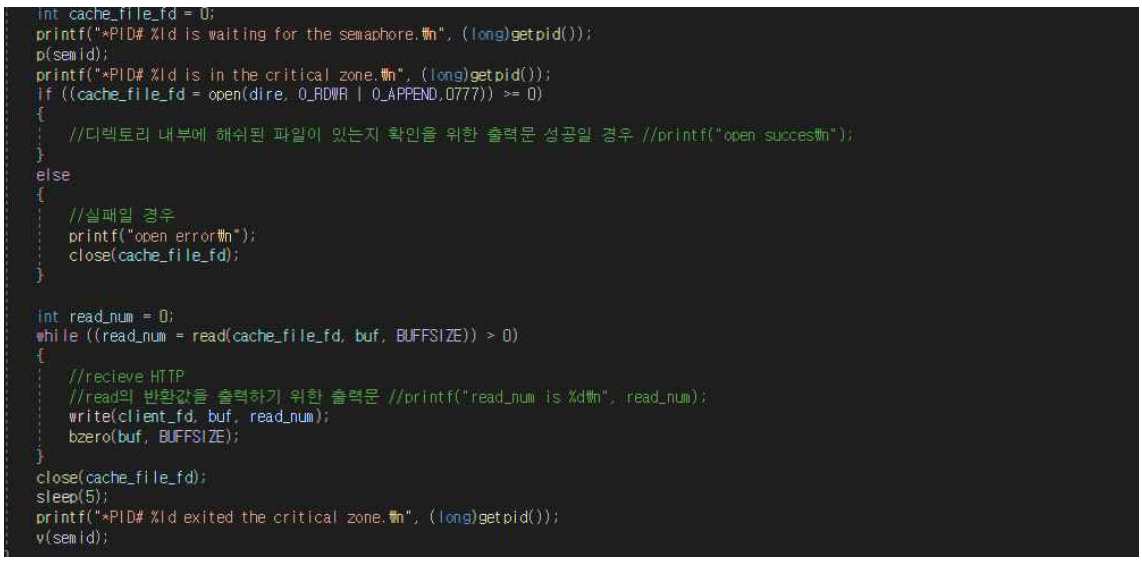
이러한 세마포어를 사용하는 부분은 다음 아래과 같다.

1. discrimination 함수 사이에서 사용하며 이유는 discrimination 함수의 경우 Hit과 Miss를 로그 파일에 저장하는 용도의 함수이므로 이는 크리티컬 섹션이다.



```
printf("*PID# %ld is waiting for the semaphore.₩n", (long)getpid());
p(semid);
printf("*PID# %ld is in the critical zone.₩n", (long)getpid());
if (discrimination(first_HS_URL, second_HS_URL, URL) == 1)//디렉토리와 파일을 위해 나눠놓은 문자열을 인자로 받는 discrimination 함수 실행
    misscount += 1;//함수의 반환값이 1이면 misscount증가
else
    hitcount += 1;// 함수의 반환값이 2이면 hitcount 증가

sleep(5);
printf("*PID# %ld exited the critical zone.₩n", (long)getpid());
v(semid);
```

2.이 경우는 Hit일 경우 캐시 파일에 접근하여 데이터를 브라우져에 띄우는 방식으로 이 또한 크리티컬 섹션의 부분들이 있다.



```
int cache_file_fd = 0;
printf("*PID# %ld is waiting for the semaphore.₩n", (long)getpid());
p(semid);
printf("*PID# %ld is in the critical zone.₩n", (long)getpid());
if ((cache_file_fd = open(dire, O_RDWR | O_APPEND,0777)) >= 0)
{
    //디렉토리 내부에 해쉬된 파일이 있는지 확인을 위한 출력문 성공일 경우 //printf("open succes₩n");
}
else
{
    //실패일 경우
    printf("open error₩n");
    close(cache_file_fd);
}

int read_num = 0;
while ((read_num = read(cache_file_fd, buf, BUFFSIZE)) > 0)
{
    //recieve HTTP
    //read의 반환값을 출력하기 위한 출력문 //printf("read_num is %d₩n", read_num);
    write(client_fd, buf, read_num);
    bzero(buf, BUFFSIZE);
}
close(cache_file_fd);
sleep(5);
printf("*PID# %ld exited the critical zone.₩n", (long)getpid());
v(semid);
```

3. 이 경우는 Miss일 경우임으로 웹 서버와 통신을 가능케 하고 리스폰스를 받으면 이를 캐시에 저장하는 과정을 거쳐야 하므로 이 또한 크리티컬 섹션이다.

```
    printf("*PID# %ld is waiting for the semaphore.₩n", (long)getpid());
    p(semid);
    printf("*PID# %ld is in the critical zone.₩n", (long)getpid());
    if ((cache_file_fd = open(dire, O_RDWR | O_APPEND,0777)) >= 0)
    {
        //캐시파일의 파일 오픈이 되는지 확인을 위한 출력문 //printf("open succes₩n");
        close(cache_file_fd);
    }
    else
    {
        printf("open error₩n");
        close(cache_file_fd);
    }
    //핸들러가 실행할 때 대기상태를 벗어나기위한 부분
    struct sigaction act, oact;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags &= ~SA_RESTART;
    sigaction(SIGALRM, &act, &oact);

    write(server_fd, buf, strlen(buf));

    alarm(15);//알람 15초 설정

    bzero(buf, BUFFSIZE);
    int read_num = 0;
    int read_num_sum=0;
    int check_num = 0;
    int Content_Length_int = 0;
    int Content_Length_before_entity_body = 0;
    bool chunck_check = false;
    int newline_count = 0;
```

```
        write(client_fd, buf, read_num);
        pFile = fopen(dire,"ab");
        fwrite(buf,1, read_num,pFile);
        fclose(pFile);
        bzero(buf, BUFFSIZE);
        if ((Content_Length_int + Content_Length_before_entity_body+ newline_count <= read_num_sum)&&(chunck_check==false))
        {

            /*
            //추출한 read_num의 길이의 합이 엔티티 바디 길이 포함 해더 길이 포함과 같은지 확인
            printf("newline_count is %d₩n", newline_count);
            printf("Content_Length_int is %d ₩n", Content_Length_int);
            printf("Content_Length_before_entity_body is %d ₩n", Content_Length_before_entity_body);
            printf("read_num_sum is %d₩n", read_num_sum);
            printf("Content_Length_int + Content_Length_before_entity_body+ newline_count is %d ₩n", newline_count+ Content_Length_int + Content_Length_before_entity_body);
            */

            alarm(0);// content length를 포함하여 content length를 알수 있을 때 read_with_time함수가 기다리는 값보다 더 빨리 종료 되게 함으로
                    // 알람을 0으로 만듬
            close(server_fd);
            break;
        }
        alarm(15);// read를 다시실행 할 경우를 대비하여 알람 15초 설정 아는 while값을 벗어나면 해제 된다.
    }
    //while 값을 종료한 지점의 read_num 값이 0인지 확인 //printf("while end point read_num is %d₩n", read_num);
    if (read_num == 0)
    {
        //read가 안전하게 끝나면 알람 해제
        alarm(0);
        close(server_fd);
    }
    else if (read_num == -1)
    {
        //웹서버에서 read를 실패할 경우 no reasponse
        //printf("while end point read_num is %d₩n", read_num);
        close(server_fd);
    }
    close(server_fd);
    sleep(5);
    printf("*PID# %ld exited the critical zone.₩n", (long)getpid());
    v(semid);

}
```

4. 이 경우는 서브 프로세스를 종료할 때의 부분으로 log파일에 종료문을 기록하여야 하고 이 또한 크리티컬 섹션으로 볼 수 있다.

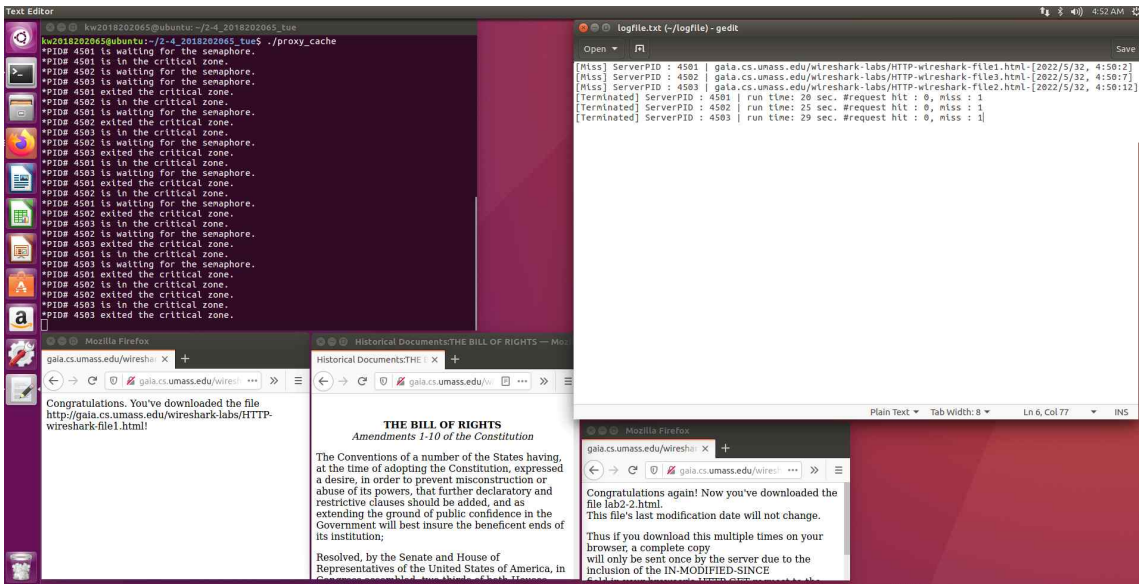
```
    }
    time(&end);
    int result = 0;
    result = (int)(end - start);
    sleep(1);
    printf("+PID# %ld is waiting for the semaphore.₩n", (long)getpid());
    p(semid);
    printf("+PID# %ld is in the critical zone.₩n", (long)getpid());
    bye(subprocess_hitcount, subprocess_misscount, result);// 서브 프로세스의 종료문을 출력하기위한 함수
    sleep(5);
    printf("+PID# %ld exited the critical zone.₩n", (long)getpid());
    v(semid);
```

총 4개의 크리티컬 섹션이 있으며 이러한 구현에 관한 결과를 다음 문항에서 설명하고자 한다.

- 실습 결과

-실습 방법 (시나리오)
크리티컬 존에 대한 순서를 다음과 같이 나열 한다.
1번 : Hit or miss를 판별해 로그파일에 저장하는 크리티컬 존
2번 : 캐시에 접근하여 브라우저에 캐시 데이터를 전송하는 크리티컬 존(Hit, Miss의 경우를 통합해서 지칭하겠다.)
3번 : 로그파일에 브라우저에 대한 서브 프로세스의 종료문을 기록하는 크리티컬 존
이후 아래의 웹사이트에 대해
http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html
http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html
http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html
거의 동시에 차례로 접속하게 되면
가장 먼저의 PID 4501 가 1번 크리티컬 존에 들어가게 되고 이에 따라 같은 일을 하는 크리티컬 존에 대해 4502와 4503은 웨이팅 상태가 된다. 그리고 4501가 1번 크리티컬 존에 나가게 되면 4502가 1번 크리티컬 존에 들어가게 된다. 또한 이후 4501는 2번 크리티컬 존에 들어가기 전 웨이팅 상태가 된다. 이후 4502가 1번 크리티컬 존을 나가면 4503이 크리티컬 존에 들어가고 4502가 2번 크리티컬 존을 들어가기 위해 웨이팅하게 되고 4503이 1번 크리티컬존을 나가면 4501가 2번 크리티컬 존에 들어간다. 또 다시 4503이 2번 크리티컬 존에 들어가기 위해 웨이팅하게 되고 4501가 끝나면 4502가 2번 크리티컬 존에 들어가고 4501이 3번 크리티컬 존에 들어가기 위해 웨이팅한다. 4502가 끝나면 4503이 2번 크리티컬 존에 들어가고 4502가 3번 크리티컬 존에 들어가기전 웨이팅 하게 된다. 이후 4503이 끝나면 4501가 3번 크리티컬 존에 들어가게 되고 4503이 3번 크리티컬 존에 들어 가기전 웨이팅 하게된다. 4501가 끈나면 4502이 3번 크리티컬 존에 들어가고 끝나게 되면 4503이 3번 크리티컬 존에 들어가고 끝나게 되면 3개의 브라우저에 대한 동시접근 통제를 마치게 된다. 올바른 결과가 나오게 되는지 터미널과 로그파일의 기록정보를 확인하여 판단한다.

-결과 화면



위 시나리오와 정확히 일치 하게 터미널에 출력됨을 알 수 있다. 1개의 프로세스가 3개의 크리티컬 존에 들어가는 모습을 터미널을 통해 알 수 있다. 또한 동시 접근을 제한하여 순서에 맞게 공유 메모리에 접근하는 것을 터미널을 통해 알 수 있다.

● 고찰

이번 과제를 진행하면서 발생한 이슈는 다음과 같다. 또한 이것에 대한 고찰을 같이 서술하고자 한다.

1. 세마포어를 삭제해 주는 semctl(semid, 0, IPC_RMID, arg)를 사용하기 위한 이슈

semctl(semid, 0, IPC_RMID, arg)을 사용하기 위해서 일반적인 프로그램 같은 경우 main 함수가 끝나서 return하기 전 semctl(semid, 0, IPC_RMID, arg)을 사용하면 되지만 이번 프록시 서버 프로그램 같은 경우 종료는 SIGINT(Ctrl+C)를 통해 종료 한다. 따라서 시그널 핸들러에서 semctl(semid, 0, IPC_RMID, arg)을 사용하기 위해서는 일반적인 프로그램과는 다른 방식으로 하여야하는데 위 프로그램에서 구현한 방법은 semid와 semum을 전역변수로 선언하는 것이였다. 이렇게 하면 프로그램 시작 후 main함수에서 semid를 받아올 수 있으며 갑작스러운 SIGINT(Ctrl+C)의 발생에 있어서도 시그널 핸들러 함수에서 semctl(semid, 0, IPC_RMID, arg)가 가능하기 때문이다.

2. 자식 프로세스의 변수 공유에 대한 이슈

부모 프로세스의 fork로 인해 생성된 자식 프로세스의 경우에 있어 부모 프로세스의 변수 그리고 다른 자식 프로세의 변수 또한 공유하지 않는다. 하지만 세마포어 값은 공유를 하는 것을 알게 되어 이에 대해 의문이 들었고 알아 본 결과 컴퓨터가 여러 프로그램을 동시에 수행하는 다중 프로그래밍 시스템에서는 프로세스들간의 상호배제와 동기화를 위한 기본적인 연산이 필요하게 되고 세마포어는 여러 프로세스들에 의해 공유되는 변수로 정의된다.
그런데 이 변수는 보통의 방법으로는 액세스할 수 없고 오직 P와 V라는 연산으로만 다룰 수 있다는 것을 알게 되어 이러한 궁금증을 해결할 수 있었다.