System Programming

Assignment#3-1

08_Proxy 3-1

Professor	목 3 4 황호영 교수님
Department	컴퓨터공학과
Student ID	2012722028
Name	장 한 별
Date	2018. 05. 31

Introduction.

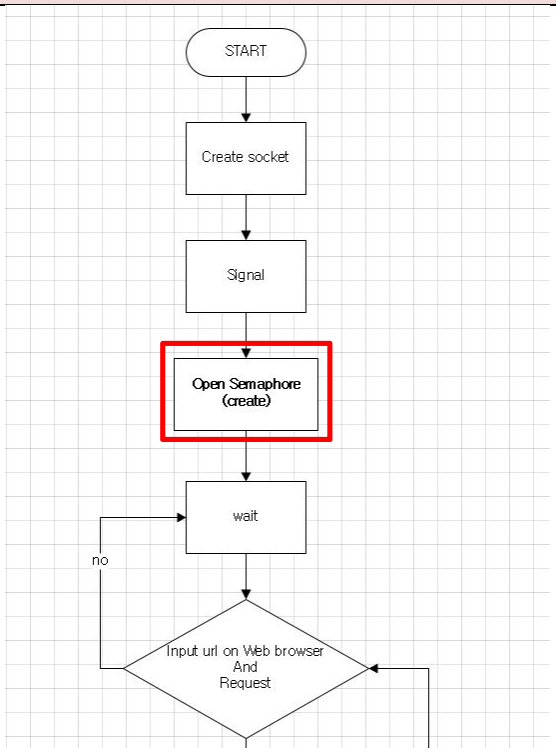
시스템 프로그래밍 강의 시간에 배운 proxy server 를 구현하는 것을 목표로 한다.

이번 과제는 Synchronize Shared Resource로, 어떤 한 process 가 critical section을 실행 중일 때, 다른 process 가 그 critical section을 접근하지 못 하게하는 Semaphore를 구현한다. Semaphore와 관련된 함수를 적절하게 활용하고, semaphore의 값에 따라 적용되는 함수가 다름을 이해한다.

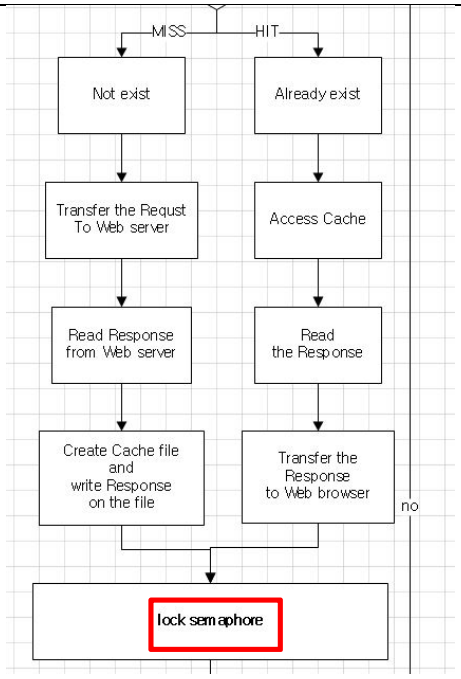
이번 과제의 critical section은 logfile.txt 에 기록하는 부분이다.

Flow chart.

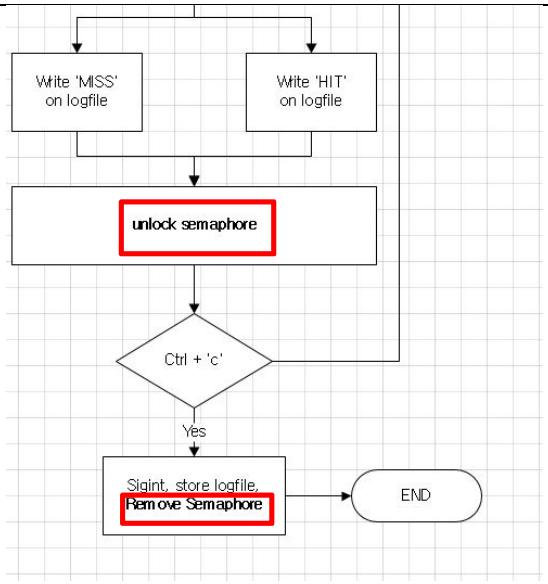
proxy_cache



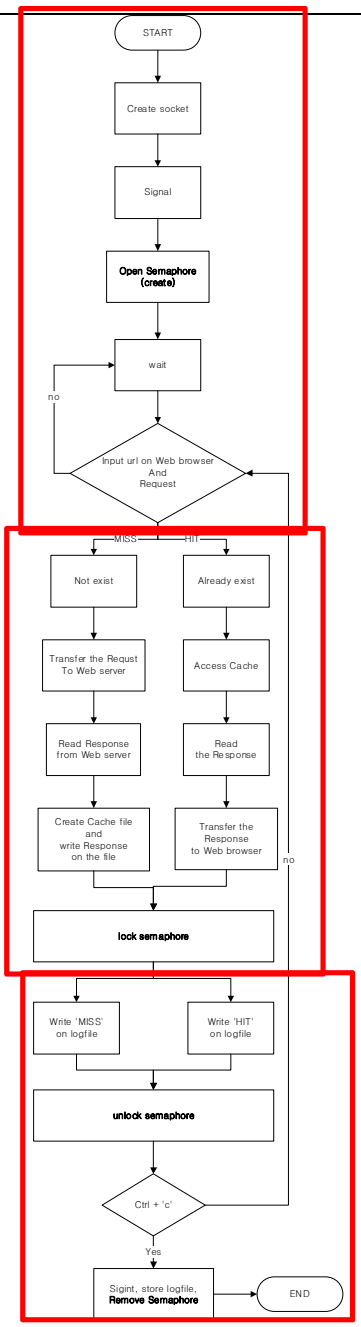
이번 과제의 flow chart는 세 부분으로 나누어 설명하도록 한다. 먼저 Semaphore를 생성하는 부분이다. 어느 한 process가 critical section에 접근하고 있을 때, 다른 process 가 이 critical section에 접근하게 되면 문제가 발생하게 되는데, 이러한 문제를 해결하기 위해 semaphore 를 활용한다.



Web browser(client)에 input 된 url로 그동안의 과정을 이용해 MISS, HIT을 판별하고 cache를 생성한다. 그 후, Cache 에 접근한 내용(MISS 나 HIT)를 logfile.txt 에 저장하게 되는데, 이 부분이 critical section 이므로 semaphore를 얻는다.(lock)



Logfile.txt 에 기록이 모두 끝났으면 critical section을 빠져나오면서 semaphore를 반납한다.(unlock) 그 후, proxy server가 완전히 종료될 때 semaphore를 제거하도록 한다.



이번 과제 의 전체 flow chart 를 한눈에 보면 위 그림과 같다.

Pseudo code.

proxy_cache
Int main(void) { Socket(); Setsockpot(); Bind(socket);

```
Listen(socket, 5);
Signal(SIGCHLD);
Signal(SIGALRM);
Signal(SIGINT);

Sem_open();

While(1)
{
    Accept();
    Read(client);

    Printf("%s", Request );
    Get url;

    Assignment #1-2;
    If(MISS)
    {
        Socket(web);
        Connect(web);
        Write(web, request);
        Alarm(10);
        While(read(web, buf))
        {
            Write(client, buf);
          Alarm(0);
        }
        Close(web);

        Fprintf(fp, response);
        Fclose(fp);

        Sem_wait( );
        Fprintf(logfile, "MISS");
        Fclose(logfile);
        Sem_post( );
    }
    Else
    {
        Read(cache, response);
        Write(client, response);

        Sem_wait( );
        Fprintf(logfile, "HIT");
        Fclose(logfile);
```

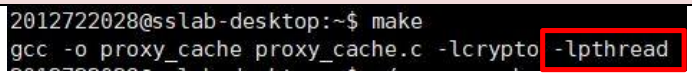
```
            Sem_wait( );
        }
        Close(client);
    }
    Close(socket);
    Sem_unlink( );
    Return 0;
}
```

위 그림은 proxy_cache 의 전체 pseudo code 이다. 먼저 Critical section에 동시에 두 개 이상의 process가 접근하지 못하도록(한 process 만 접근하도록)하는 semaphore를 생성(open 및 초기화)한다. 그후, MISS 나 HIT 가 발생시, 그 내용을 logfile.txt 에 기록하게 되는데 이 부분이 critical section 이므로 이 section에 들어가기전에 semaphore를 얻는 함수인 sem_wait( ) 함수를 호출하고, 이 section을 빠져나올 때 semaphore를 반납하는 함수인 sem_post( ) 함수를 호출하도록 한다.

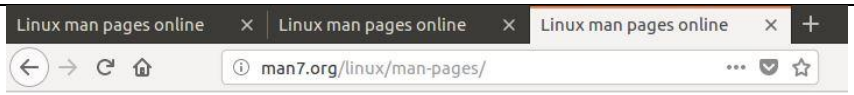
Proxy server가 ctrl+ 'c'로 종료될 때, signal을 보내게 되는데, 이 signal로 semaphore를 제거 시키는 함수인 sem_unlink( ) 함수를 호출한다.

Result.

3-1.(1)

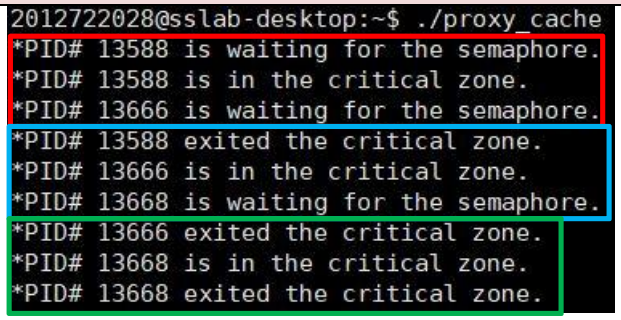


Semaphore관련 함수가 있는 코드를 컴파일 하기 위해 위 그림과 같이 '-lpthread' 옵션을 추가한다. '-pthread' 를 추가해도 무방하다.



컴파일이 완료되었다면, semaphore가 제대로 작동하는지 확인하기 위해 3개의 filefox를 실행시켜 동시에(짧은 시간내에) 같은 url('man7.org/linux/man-pages')을 입력한다.

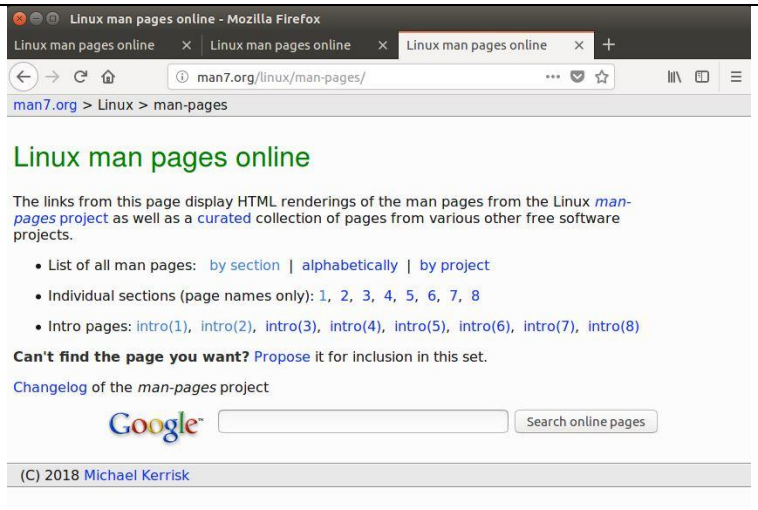
3-1.(2)



위 그림 중 빨간색 네모칸을 살펴보도록 한다. Process ID 가 13588 process 가 semaphore를 얻고, critical section으로 접근했다는 것을 확인할 수 있다. 아직 그 process 의 작업이 끝나지 않았는데 13666 process 가 그 critical section으로 접근하려 하자, 그것을 막고 기다리게 한다.

다음으로 파란색 네모칸을 확인해보면, 13588 이 critical section을 빠져나온 것을 확인할 수 있다. 그 후, 기다리고 있던 13666이 critical section으로 접근한다. 또 다른 process인 13668이 critical section을 접근하려 하자 이를 막고, 기다리게 한 것을 확인할 수 있다.

마지막으로 초록색 네모칸을 확인해보면, 1366 이 critical section을 빠져나오고, 13668이 critical section에 접근했음을 확인할 수 있다. 이후 더 이상 기다리는 process가 없으니 13668은 작업이 끝나면 그대로 semaphore를 반납하고 빠져나온다.



Web browser를 확인해보면 세 개의 browser가 모두 올바르게 출력되었음을 확인할 수 있다.

3-1.(3)

```
^C2012722028@sslab-desktop:~$ cat ~/logfile/logfile.txt
[Miss]man7.org/linux/man-pages-[2018/5/31, 21:36:17]
[Hit]f3a/fe9a64ddd001d5bff149d718dab6cb68a43f9-[2018/5/31, 21:36:22]
[Hit]man7.org/linux/man-pages
[Hit]f3a/fe9a64ddd001d5bff149d718dab6cb68a43f9-[2018/5/31, 21:36:23]
[Hit]man7.org/linux/man-pages
**SERVER** [Terminated] run time: 21 sec. #sub process: 6
```

위 그림은 Logfile.txt 를 확인한 결과 화면이다. Semaphore를 사용하지 않았다면, 가장 마지막으로 접근(가장 최근에 접근)한 process의 정보가 기록된다. Semaphore 를 사용해 이와 같은 문제를 막을 수 있었고, 위 그림을 확인하여 MISS, HIT, HIT 순으로 올바르게 기록되었음을 확인할 수 있다.

```
2012722028@sslab-desktop:~$ ls
cache  examples.desktop  logfile  Makefile  proxy_cache  proxy_cache.c
2012722028@sslab-desktop:~$ tree ~/cache
/home/2012722028/cache
└── f3a
    └── fe9a64ddd001d5bff149d718dab6cb68a43f9

1 directory, 1 file
2012722028@sslab-desktop:~$
```

Cache 역시 올바르게 생성되었음을 위 그림을 통해 확인할 수 있다.

Conclusion.

이번 과제에선 두 가지 이상의 process 가 critical section에 접근하려 할 때, 이를 처리해주는 semaphore 를 구현했다. Semaphore는 이미 한 process가 critical section에 접근하고 있을 시, 새로운 process 가 이 critical section을 접근 하려하면, 이를 막고 기다리게 한다. 그 후, 기존의 process 가 critical section을 빠져나오면 semaphore를 반납하고, 새로운 process가 critical section을 접근하게 된다. 마치 옷가게에서 탈의실을 들어갈 때, 옷을 갈아입으려고 번호표를 받고 탈의실을 들어갔다 나왔다하는 모습과 비슷하다.

Semaphore와 관련된 함수들을 잘 활용한다면, 이번 과제는 비교적 쉬운 난이도에 속했을 것이다. Semaphore를 생성(open 혹은 초기화)해주는 함수인 sem_open( ) 함수, 한 process가 critical section에 접근할 때, semaphore를 얻으려고 사용하는 함수인 sem_wait( ) 함수, critical section을 빠져나올 때 semaphore를 반납하려고 사용하는 함수인 sem_post( ) 함수, 그리고 마지막으로 semaphore를 제거하는 함수인 sem_unlink( )함수가 그러하다. Semaphore는 main함수 초반에 생성하여 critical section에 접근할 때 사용하게 되는데 이번 과제의 critical section은 MISS 나 HIT 정보를 logfile.txt에 기록하는 부분이다. 만약 semaphore를 사용하지 않는다면, 가장 마지막으로 접근한(가장 최근에 접근한) process 의 결과에 따라 logfile.txt 에 기록되었겠지만, semaphore를 사용하면 여러 개의 process 를 번호표를 주어 하나씩 하나씩 처리할 수 있도록 도와준다.

이번 과제에서 쉽게 해결되지 않았던 부분은 sem_unlink( ) 함수의 위치였다. 기존에 실행해서 생성된 semaphore 가 남아있었던 탓인지, 아무리 다시 시도해도 web browser에 web server로부터 받아온 response 조차 write 되지 않아 무척이나 당황했었다. 그래서 sem_unlink( )함수의 위치를 sem_open( )함수를 호출 하기전에 사용해봤고, 그 이후로 작동이 잘 되어서 원래대로 다시 위치를 돌려놨더니, 이론적으로 완벽한 semphore의 전개가 완성되었다. 또한 이번 과제에선 이전 과제에서 문제가 되었던 메모리 문제를 해결할 수 있었는데, 이전 과제부터 시작해서 다시 하나하나 차근차근 코딩해 보는 방법으로 찾을 수 있었다. 원인은 fclose 를 두 연속 사용한 것이었다. 이것을 해결하니 메모리 문제도 말끔하게 해결했다. 3-1 과제를 끝마치는 것으로 드디어 시스템 프로그래밍의 끝이 보인다. 끝까지 포기하지않도록 최선을 다할 것이다.