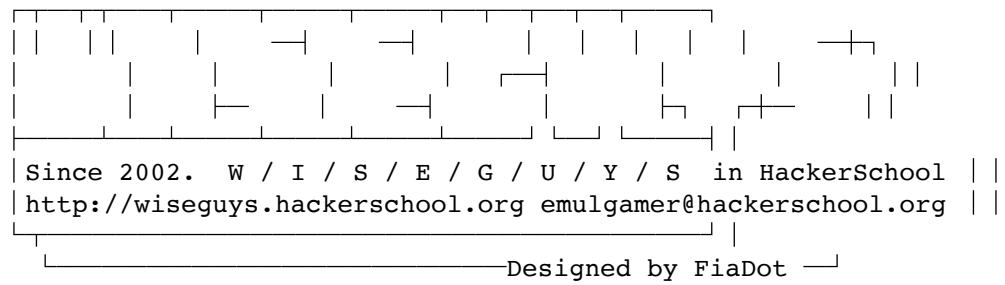


—[Document Infomation] ———————————————————

:: Title :: 세마포를 정복하자.
:: Date :: 2004. 3. 10.
:: Type :: translation
:: Author :: ChuRack
:: Contact:: E-Mail(churack@bluetooth.chonnam.ac.kr)

———————————————————————————————

—[Notice]———————————————————————————

본 문서의 잘못된 부분이나 지적이나 추가하고 싶은 내용은
저자에게 메일을 보내기 바랍니다.

———————————————————————————————

—[Index] ———————————————————————————

———————————————————————————————

—[0x00 시작하면서..]———————————————————

이번에는 해킹에 관한 기술이 아니라, 세마포의 개념에 대해서 설명을
해 보았다. 개인적으로 OS 수업시, 가장 재미있었던 부분이며, OS의
핵심기술 중 하나인 세마포. 세마포를 printf쓰듯이 쓸수만 있다면 연
봉 5000만원도 받을 수 있다는 유언비어가 나돌 정도의 중요한 내용이
다. 그럼 세마포에 대해서 알아보자.

—[0x01 이런! 내 월급~~]———————————————————

해쿨홈페이지를 관리하면서 한달에 50만원씩 받는 A씨. 월급 들어오는

날 기분을 내 옷을 사기 위해 통장에 있는 10만원 중, 5만원을 인출하였다. 인출하는 순간 월급이 입금되었다는 문자를 받았다. A씨는 기쁜 마음에 잔고를 확인해 보았다. 그런데 잔고에 5만원이 있는게 아닌가! 이게 뭔 일이지?

어찌 된 상황일까.
돈을 인출하는 상황 :
  1) 메모리로부터 잔고의 내용을 레지스터에 전송한다.
  2) CPU는 레지스터 내용에서 5만원을 뺀다.
  3) 레지스터의 내용을 메모리로 전송한다.
돈을 입금하는 상황 :
  4) 메모리로부터 잔고의 내용을 레지스터에 전송한다.
  5) CPU는 레지스터 내용에서 50만원을 더한다.
  6) 레지스터의 내용을 메모리로 전송한다.

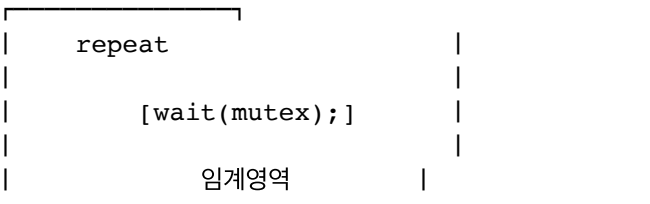
이런 활동들이 일어 났을 것이다. 그런데 왜 50만원이 사라졌지?
멀티프로세싱이 가능한 컴퓨터. 이런 순서로 동작하면 어떻게 될까?
  1) -> 4) -> 5) -> 6) -> 2) -> 3)
50만원의 행방이 이리하여 날라가버리는 것이다. 이렇게 동시에 여러개의 프로세스가 동일한 자료를 접근하여 조작하고, 실행 결과가 접근하는 특별한 순서에 의존하는 것을 경합상태(race counter)라 말한다. 말도 안된다고? 만일 제어를 해 주지 않는 다면 이런 상황이 많이 발생할 수 있다.

—[0x02 세마포란?]————————————————————————

그럼 오늘의 이슈 세마포는 무엇일까? 간단히 말해 세마포는 같은 자원을 동시에 접속하지 못하도록 제어해 주는 것이다. 그럼 세마포는 어떻게 동작을 하는 것일까? 세마포는 우선 임계영역과 잔류영역을 구분함으로써 상호배제를 구현한다. 여기에서 상호배제란 하나의 자원에 여러개의 프로세스가 동시 접속하는 것을 막는 것을 의미한다. 임계영역이란 프로세스가 공유자원에 접근하는 부분을 의미하며, 잔류영역이란 공유자원을 사용하고 O/S에 자원에 돌려주는 부분을 의미한다. 즉 경합상태가 일어날 가능성이 있는 부분인 임계영역에 한 프로세스가 들어갔을 경우, 다른 프로세스들은 임계영역에 들어가지 못하도록 막는 것이다.
세마포의 다음 구성으로는 wait(S)와 signal(S)를 들 수 있다. wait(S)는 어떤 한 프로세스가 임계영역에 있을 경우, 다른 프로세스들이 임계영역에 접근하지 못하고, 임계영역에 있는 프로세스가 잔류영역으로 나올때까지 기다리도록 하는 연산이다. signal(S)는 임계영역에 있던 프로세스가 잔류영역으로 나오면서 자신이 임계영역에서 나왔다는 것을 다른 프로세스에게 알려주는 연산을 한다. 또한 세마포는 mutex(mutual exclusion)라는 공동 세마포를 공유하며 동시 접속이 가능한 자원의 수만큼을 표기한다.(보통 1) 예를 들어 프린터가 하나가 프린터에 접근할 수 있는 프로세스의 수는 1개가 된다. 그럼 프린터가 두개 있는 상황은 어떻게 될까? 그럼 프로세스가 접근할 수 있는 프린터의 수는 2개이므로 두개의 프로세스가 동시에 접근할 수 있다. 그러므로 mutex는 2가 된다고 할 수 있다. 빈약한 예가 될 지 모르지만, 공유할 수 있는 자원의 수라고 생각하면 쉽게 이해할 수 있다고 생각한다.

—[0x03 그럼 세마포는 어떻게 생겼을까?]————————————————

세마포의 구성요소는 위에서 살펴 보았다. 그런데 도대체 이놈의 구성요소가지고는 어떻게 동작하는지 이해가 안 될것이다. 그럼 세마포 이놈이 어떻게 생겼는지 보자.



```
|                              |
|        [signal(mutex);]      |
|                              |
|             잔류영역         |
|                              |
|    until false;              |
└──────────────┘
```

이런 식으로 구성되어 있다고 보면 될 것이다.
즉 임계영역에 들어가기전 wait 연산을 통하여 임계영역에 어떠한 프로세스가 있는지 확인한 후, 임계영역에 있는 프로세스가 없을 시, 자신이 임계영역을 들어가고 임계영역에 있는 프로세스가 있을 시, 임계영역에 있는 프로세스가 나올때까지 기다린다. 또한 임계영역에서 빠져나온 프로세스는 잔류영역에 들어가기 전, signal연산을 통하여 자신이 임계영역에서 나왔다는 것을 임계영역에 들어가기를 기다리고 있는 프로세스들에게 알려준다. 그리고 잔류영역에 들어가, 나머지 연산을 수행하는 것이다. 세마포는 이러한 동작을 통하여, 간단하게 경합상태를 방지해 주고 있는 것이다.

—[0x04 wait와 signal연산에 대해 알아보자]————————————

세마포가 어떻게 구성되어 있는지는 알았다. 그런데 어떠한 방식으로 wait와 signal연산이 동작하는 것일까? wait와 signal의 구현에 대해 알아보자.

```
┌──────────────────────────┐
|    wait(S) : S.value := S.value -1;            |
|        if S.value < 0                          |
|            then begin                         |
|                add this process to S.L;        |
|                block;                          |
|            end;                               |
|                                                |
|    signal(S) : S.value := S.value + 1;         |
|        if S.value <= 0                         |
|            then begin                         |
|                remove a process P from S.L;    |
|                wakeup(P);                      |
|            end;                               |
└──────────────────────────┘
```

우선 wait연산을 살펴보자. wait연산을 처음에 들어온 인자값인 S의 value값을 1 감소시킨다. 이것을 해석보자. 우선 위에서 나온 세마포 구성 그림에서 wait()연산에 들어간 인자값은 무엇이었나? 기억이 안 나는 사람은 30초동안 손들고 반성하시길...——^ 잠시 위로 올라가보자. wait(mutex)라고 쓰여 있지 않은가. 즉 임계영역에 들어가기 전 공유자원을 자신이 사용하고 싶기 때문에, mutex를 1감소시키는 것이다. 그 후, S.value(즉 mutex)값이 0이하인지 살펴본다. 만약에 공유자원을 아무도 사용하지 않았다면 1에서 1을 뺀(mutex 초기값이 1이라고 가정하였을 시) 값은 0이된다. 즉 해당되지 않는 것이다. 이 경우에는 프로세스는 wait문을 빠져나가 임계영역으로 진입하게 되는 것이다. 그럼 다른 프로세스가 공유자원을 사용하고 있어서, mutex의 값이 0이었으며 1을 뺀 후, -1이 되었을 상황을 가정해 보자. 이 경우에는 if문에 true가 된다. 그럼 이 프로세스를 S.L에 추가시킨후, 블록시킨다. 이게 무슨 말인지 살펴보자면, 다른 프로세스가 공유자원을 사용하고 있기 때문에 자신은 임계영역에 들어갈 수 없다. 그러므로 대기하고 있어야 한다. 이 대기하는 방법이 여러가지가 있지만 가장 쉬운것이 폴링이라는 방법으로 무한루프를 돌면서, 다른 프로세스가 임계영역에서 나왔는지 계속 물어보는 것이다. 간단하게 구현을 하자면

```
    the begin
        while(1)
            if S.value >= 0
                then begin
                    break;
                end;
    end;
```

이 방법이 될 것이다. 프로세스는 멈추지 않고 S.value를 계속 확인을 하며, 다른 프로세스가 임계영역에서 나왔냐고 계속 물어볼 것이다. 그러나 이 방법의 경우 CPU의 오버헤드를 너무 많이 차지하게 된다. 그래서 여기에서는 S.L라는 대기큐(queue)에 이 프로세스를 집어 넣은 후, 프로세스를 block을 시켜버리는 것이다. 이렇게 block된 프로세스는 깨워줄때까지, 즉 다른 프로세스가 임계영역에서 나올때까지 계속 쉬고 있게 된다.
그럼 signal연산을 살펴보자. signal연산을 S.value를 우선 1 증가시킨다. 자신이 공유자원을 O/S에 반환하겠다는 의미가 된다. 그 후 S.value가 0이하인지 살펴본다. 만약에 0이하라면 자신이 임계영역에 있는 동안 다른 프로세스가 임계영역 접근에 시도하였다가 세마포에 걸려 접근하지 못하고 대기큐에서 대기하고 있는 상태라는 것을 의미한다. 그럼 이 프로세스는 대기큐에서 프로세스를 P를 끄집어 내고 wakeup(P)연산을 통해 block되어 있는 프로세스 P를 깨우는 것이다. 그리고 자신은 잔류영역에 들어가게 되는 것이고 깨어난 P는 이제서야 임계영역에 접근하여 연산을 시작하는 것이다.

—[0x05 세마포로 인해 발생하는 문제점 DeadLock]————————————

에잉? 뭔 헛소리냐고. 이 좋은 세마포에 무슨 문제점이 있냐고? 헐헐. 세마포라 할 지라도 경합상태해결하는 모든 방안이 가진 문제점을 똑같이 가지고 있다. 그것은 바로 교착상태(DeadLock)이다. 엥? 세마포가 교착상태를 해결하는 것이 아니냐고? 물론 아니다. 저자도 가끔 헷갈리는 경우가 있는데, DeadLock을 해결하는 것이 세마포가 아니라, 세마포로 인해 DeadLock이 발생하는 것이다. 본 저자 저번 리눅스 프로그래밍 스터디 시간에 했던 "세마포는 교착상태를 해결하기 위해 사용되요"라고 했던 뻘소리 이해해주길 바란다. 가끔 헷갈린다...——;; 그 수업 끝나고 생각해보니 잘못 말했다는 것이 생각났다. 죄송합니다. 그럼 어떻게 DeadLock이 발생하는 것일까? 우선 두개의 프로세스 P1과 P2가 공유자원 M1과 M2를 필요하다고 해보자, 그런데 P1은 공유자원 M1을 소유하고 있고 프로세스 P2는 공유자원 M2를 소유하고 있다고 해보자. 이 상황에서 P1은 M2에 대한 세마포의 임계영역에 진입하려고 할때, 이미 임계영역에 P2가 진입해 있기 때문에 진입하지 못하고, 대기상태에 있게 될 것이다. 이와 마찬가지로 P2 또한 M1의 임계영역에 진입하지 못하고 대기상태에 있게 된다. 이리하여 P1과 P2프로세스 모두 자원을 획득하지 못하였기 때문에, 실행되지 않고 계속 block되어 있는 것이다. 이게 DeadLock으로써, 세마포와 같은 경합상태를 해결하기 위해 만들어진 알고리즘으로 인해, 발생하는 문제점이 되겠다. 이 DeadLock을 해결하기 위한 방법 및 알고리즘은 많이 나왔으나, DeadLock의 실제 발생빈도는 극히 드물기 때문에, 많은 운영체제들이 DeadLock 해결책을 실제 O/S에 구현하지는 않았다고 한다.

—[0x06 마치며]————————————————————————————

 해킹에 관련된 내용은 아니지만, 재미있게 읽어 주셨으면 감사하겠습니다. 특히 요즘에 몸이 안 좋아서 집에서 요양한다는 핑계로 공부도 해쿨 활동도 뜸하고 지내고 있었던거 죄송합니다. 이 문서를 시작으로 이제 다시 열심히 해보겠습니다.