운영체제 프로젝트2

미디어학과 201621025 신주연

**A. 과제 1**

1. 과제의 개요 및 목적

일반적으로 Race Condition 이란 두 개 이상의 프로세스가 공통 자원을 병행적으로(concurrently) 읽거나 쓸 때, 공용 데이터에 대한 접근이 어떤 순서에 따라 이루어졌는지에 따라 그 실행 결과가 달라지는 상황을 말한다. 이로 인하여 예측하지 못한 결과가 발생할 수도 있으며, 이를 이용하여 시스템을 해킹하려고 드는 경우까지 발생한다.

이러한 문제점이 발생 가능하게 만드는 부분, 즉 중요 resource가 위치한 부분을 Critical Section이라 하고, 이 부분의 무결성이 깨지는 것을 Critical-Section Problem이라고 한다. 이 Critical-Section Problem 이 발생하는 것을 막기 위해선, 동시에 한 영역을 사용할 수 있는 process를 하나로 지정해 주어야 하고, 상호 배제의 상황(하나의 process만 Critical-Section 을 사용하고 있는 상황)에서 이 process가 더 이상 진행이 안되면 해당operation이 더 이상 진행되지 않으므로 entry이후 exit가 보장되어야 하며, starvation문제가 발생하지 않도록 해 주어야 한다.

이러한 세 가지 조건을 모두 만족하는 Critical-Section Problem의 해결 방안으로 mutex(), semaphore(), mointal() 등의peterson’s solution을 실제로 작성하여 제공해주는 라이브러리를 사용할 수 있는데, 그 중에서도 이번 과제에선 교재 예시에서의count변수에 해당하는 tot변수에 동시에 접근하는 3개의thread를 생성하여 race condition이 발생하는 상황을 만들어 보고, mutex() 라이브러리를 사용하여 이러한 상황 속에서 3개의 thread의 중요 데이터의 접근을 관리해 보려고 한다.

이를 통해 여러개의 thread가 한가지 리소스를 두고 경쟁하는 상황에서 발생 가능한 race condition을 확인하고 공유 변수를 write하는 것으로 결과 값이 달라짐을 살펴보겠다. 또한 mutex락을 사용하여 이를 방지하는 프로그램도 작성해 본 후 실행시켜 결과를 살펴보면서 실제로 Critical-Section Problem 이 해결되었는지 확인해 보도록 하겠다.

2. 프로그램 설계

먼저 race condition이 발생하는 상황의 설계는 다음과 같다. tot라는 공유 변수가 전역 변수로 존재하고 메인 함수에선 세 개의 thread를 동시에 실행시킨다. 이 3개의 thread는 각각 compute의 함수를 실행하는데 이 함수의 내용은 전체적으로 동일하다. 이후의 mutex락을 이용한 통제 코드에서도 하나의 compute함수로 설계하였다. 원래 는 실행 결과를 한 눈에 보기 위하여compute1의 operation은 tot++; 이고 compute2의 operation은 tot+=2; 이고 compute3의 operation은 tot-=3; 이어서 각각의 thread가 race condition없이 수행될 경우 tot=0이 나와야 하는 프로그램을 구현하였지만(이후에 설명에 사용할 예정)실행된 process의 결과가 나오므로 이는 크게 주요하게 생각하지 않았다. (두 가지 모두 첨부하여 제출하였다.) race condition통제 이후의 코드는 thread의 실행 순서만 확인해도 충분하므로 모든 thread가 comput의 함수를 수행하도록 설계하여 코드를 간단히 하였다.

또한 이 결과를 눈으로 확인하기 위해 파일 입출력 기능을 넣어서 txt파일에 log를 작성하도록 구현하였으며 파일에는 시간 순서대로 결과를 볼 수 있도록 아래 첨부된 사진과 같이 밀리sec단위까지 구현할 수 있도록 설계하였다. 더해서 thread의 정보는 각각



\*thread변수를 사용하여 1, 2, 3의 번호를 매기고, 이 역시 시간 옆에 같이 출력하도록 하여 어떤 순서에 어떤 thread가 어떠한 operation을 진행하였는지 확인 및 분석이 가능하도록 설계 하였다. 또한 파일은 여러 경우를 작성해 두고 비교해보기 위하여 파일 이름을 따로 입력 받을 수 있게 하여 여러 저장 파일의 생성을 가능하게 설계 하였다.

이와 같이 프로젝트를 설계 한 후 실행하여 파일로 출력되는 결과 값을 가지고 race condition발생 여부와 그 위치를 파악해 보고 분석 해 보겠다. 또한 이를 통제하는 mutex락을 추가한 코드에서도 마찬가지로 파일로 로그를 작성하여 올바른 시행이 되었는지를 확인해 보겠다.

3. 프로그램의 주요부분 설명

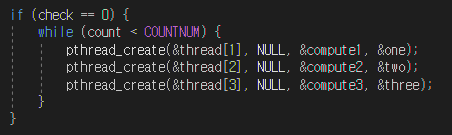
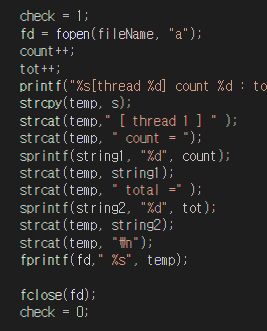
-동기화 미 작동 시

전체적인 흐름은 main함수의 흐름과 같다. 가장 먼저 log를 작성할 파일의 이름을 입력 받고 파일을 생성한다. test.txt라고 입력 후 enter시 test.txt파일이 생성되고 그 안에 log내용이 출력된다. 이 파일은 후에 권한을 “a”로 열어 로그를 출력할 때 마다 이어서 작성하게 된다. 파일의 생성이 완료되면 파일이 생성되었음을 화면에 출력하여 나타낸다. 이후로 pthread\_create함수를 이용하여 thread3개를 생성하고 각 thread가 실행하는 함수 속에는 count와 tot를 증가시키는 operation과 이를 log메시지로 출력하는 기능이 들어있다.

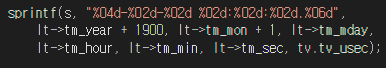
compute 함수 세 가지를 살펴보자. 이 함수에서는 파일을 열고 시간과 해당 함수를 실행시키고 있는 thread의 번호와 몇 번째 operation수행중인지를 나타내는 count변수의 값과 최종적인 tot값을 나타낸다. 가장 중요하게 보아야 할 것은 check변수인데, 이 변수의 용도는 하나의 thread가 모든 상황을 통제하는 starvation과 같은 상황을 막기 위함이다.

check변수의 초기값은 0으로 설정해 두고 함수가 실행될 때 1로 바뀌며, 종료될 때 다시 0이 되는 형태를 띄게 되는데 메인 함수에선 다음과 같이 이 check값이 0인 경우에만 thread가 실행되도록 진행하였다. 이 에 대한 설명은 4번에서 다시 진행하도록 하겠다.

이는 Critical-Section Problem발생 이유 중 하나이지만 필자는 이로 인한 Critical-Section Problem보다 동시에 한 영역을 쓸 수 있는 process가 하나인 경우가 아닐 때로 인해 발생하는 Critical-Section Problem에 대해 더 자세히 보고 싶었기에 각 함수의 처음과 끝에 check 변수를 두어 이를 조절하였다. 위의 두 사진에서처럼 각각의 thread를 동일한 비율로 실행시키기 위해 노력하였으나, 사실상 이는 10만번의 수행을 전제조건으로 하고 난 이후에는 크게 사용되지 않았다. 첫 번째 사진은 메인 함수의 일부 코드이고 두 번째 사진은 thread가 실행하는 함수의 일부이다.



또한 다음 첨부한 사진은 밀리초 단위까지의 시간을 나타내기 위한 코드인데 이를 위해 구조체 변수와 time.h헤더파일을 사용하였다.

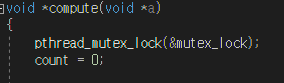


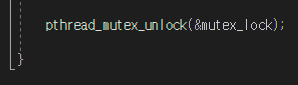
이어서 여러 thread가 count값을 증가시키면서 count값이 1000이 넘어가는 경우까지 수행되는 일이 생겨서 이를 조절하기 위해 아래와 같은 예외처리를 추가하였다.



- 동기화 작동 시

mutex락을 실행하였을 때의 코드는 일부 수정하였다. 이전에 통제하지 않은 수행에서는 각각의 thread의 다른 operation으로 진행하여 다양한 수행 결과를 확인했지만 mutex락을 설정한 후에는 이것이 크게 사용되지 않으므로 하나의 함수를 이용하여 세 개의 thread를 작동시켜 코드를 간편화 하고 유지보수를 강화하였으며 여러 thread가 하나의 함수를 수행함을 확실히 하였다. 전반적인 코드의 내용은 동일하며

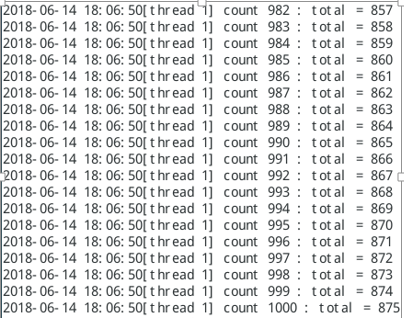




함수의 시작과 끝에 lock과 unlock을 하여 다른 thread가 현 thread의 진행 상황을 기다리게 하고 침범하지 못하게 하는 코드를 추가하였다. (이외에도 메인 함수에init과 destroy 함수를 추가하였다.)

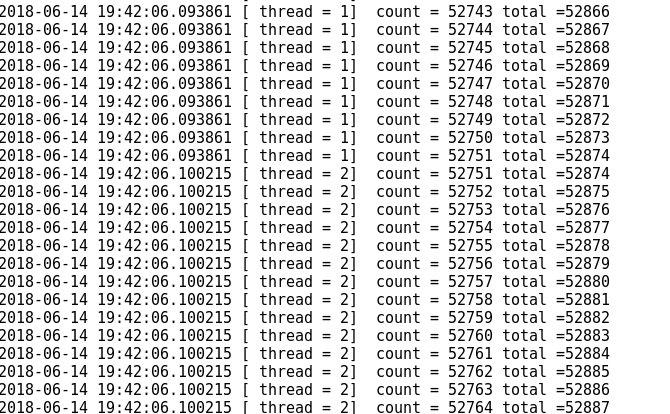
4. 프로그램 실행조건 및 상황설명

이전의 gcc명령어와는 다르게 본 프로젝트는 여러 thread를 이용하는 만큼 다른 명령어를 사용하였다. “gcc –o a cnt.c –lpthread”라는 명령어를 사용하여 a를 수행하였다.

프로그램의 실행 결과를 예측 해 보았을 때 프로그램의 실행 조건이mutex락을 사용하지 않았을 경우일 때엔 thread의 순서가 뒤죽박죽일 것을 예상했다. 동시에 세 개의 thread가 생성되고 이 세 개의 thread가 찰나의 틈을 차이로 같은 변수를 두고 operation을 수행할 것이기 때문이다. 그러나 처음 수행 시 예상 외로 이상한 결과가 나왔었다. 

위의 사진처럼 하나의 thread가 독점하여 모든 계산을 수행하고 나머지 두 개의 thread는 수행되지 않는 즉starvation문제가 발생하였다. 이를 해결하기 위해 앞에서 언급하였던 check변수를 사용하였고 이후에 아래의 사진과 같이 보다 정상적인 결과를 얻을 수 있었다. check변수를 사용하지 않을 경우 처음엔 위와 같이 하나의 thread만 수행되는 일이 거의 대부분이었고 10만번의 수행을 하는 경우 나중이 되어서 가끔 2, 3번의 thread도 번갈아 가면서 수행 되곤 했다. (위의 수행 결과는 thread별로 다른 operation을 주었을 때의 결과이다.) 그러나 check변수를 사용하여 통제하면

처음 실행순간부터 마지막 실행순간까지 모든 thread가 starvation 현상 없이 수행된다.



위의 상황은 공유변수인 count와total을 3개의 thread가 동시에 수행하고 있는 결과이고 수행 시간을 보면 굉장히 찰나의 순간임을 알 수 있다. 위의 사진에서는 특히 3번 thread가 많이 수행되어 원래 0으로 초기화 되어있던 total값이 음수의 값까지 내려온 일부 상황이다. 자세한 수행 결과는 다음 결과 설명에서 다루도록 하겠다.

뮤텍스락을 걸어놓은 상태에서는 한thread의 수행이 모두 완료된 상태에서 다음 thread의

수행이 이루어지므로, thread 1 의 while문이 모두 수행된 후 다음 thread2의 while문이 수행되고 마지막으로 thread3 의 while문이 수행될 것을 예측하였고 결과적으로 이 예측이 맞음이 증명되었다.

간단한 실행을 위해

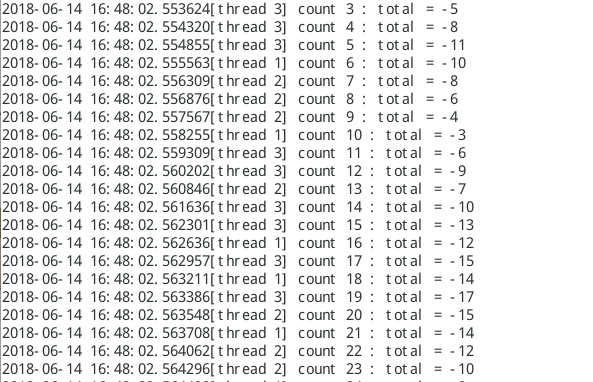


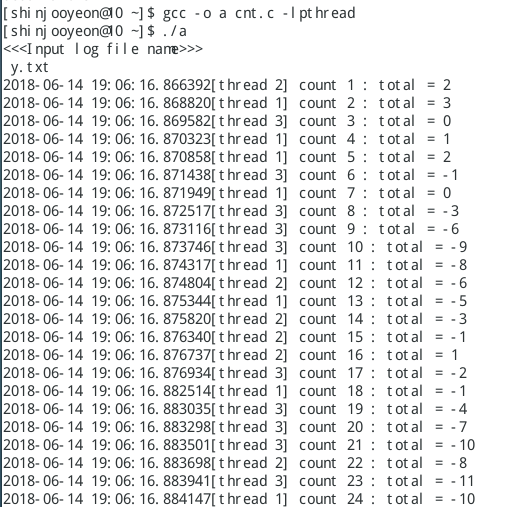
이와 같이 1000개의 reference를 이용하여 돌려본 결과 마지막 결과 값이 아래와 같이 나왔다. 또한 thread가 순서대로 실행되었다. 이 또한 다음 결과 설명에서 자세히 다루도록 하자.



5. 프로그램 실행결과 스크린샷 및 설명

16시에 수행한 1차 수행결과와 19시에 수행한 2차 수행 결과를 비교하면,

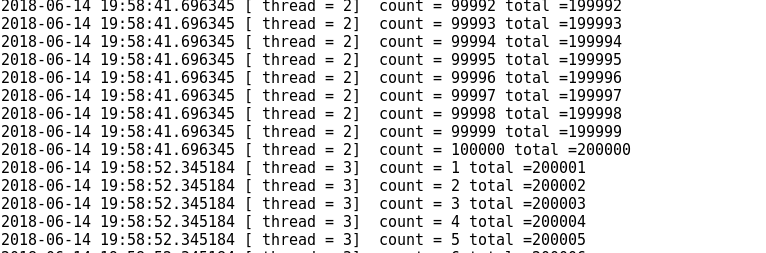
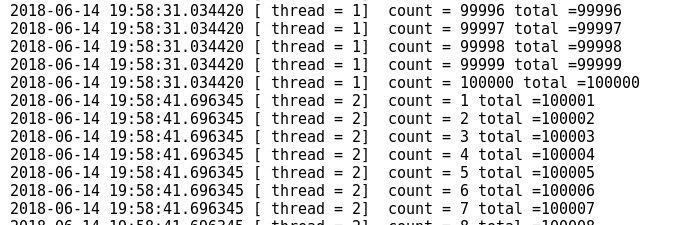


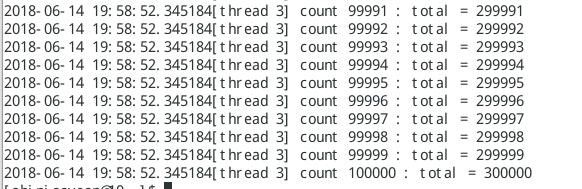
이처럼 시작하는 thread부터 다름을 알 수 있다. 위의 두 결과는 최종 실행본이 아닌 thread별로 operation을 다르게 시행했던 실행인데 count별로 exit도 사용해서 락이 많이 발생하였다. 그러나 한눈에 결과값이 잘 보인다. 이로 인하여 count가 진행 될 때마다 바뀌는 값 역시 달라지고 1차 2차 수행 값은 거의 모든 값이 다르다.

이는 process 1 수행되고 있는 중간에 다른 process2와 3이 수행되기 때문이고 실행 순서에 따라 값이 달라지는 상황이 발생한다. time shared이기 때문에 process가 concurrently하게 수행되고 이는 동시에 다른 수행이 될 수 있다는 것을 의미한다. 따라서 race condition이 발생하게 되고 이로 인하여 위와 같은 data inconsistance문제가 발생하며 위와 같이 데이터를 쓰는 문제에서는 이 상황이 심각해질 수 있다. 지금 시점에서 어떤 코드가 사용되고 있는지는 초당 만 번 정도 수행되는 process의 수행에선 알기 어려우므로 이런 상황에서 여러 개의 프로세스가 동시에 하나의 변수에 접근하는 경우 위와 같은 덮어쓰기의 문제가 발생한다.

또한 total값에 ++을 하는 간단한 연산도 사실은 메모리에서 변수에 들어있는 값을 레지스터에 옮기고 레지스터에 옮긴 값을 1 증가 시킨 후 이것을 다시 메모리의 변수에 저장하는 방식으로 CPU레벨에서 분화되기 때문에 중간에 context switching을 통해서 다른 process가 total변수에 접근하게 되면 위와 같이 순서에 따라 결과값이 달라지는 현상이 일어난다.

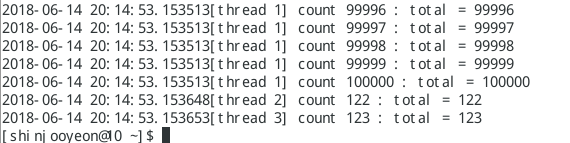
다음으로 mutex락을 적용한 경우를 살펴보자. 아래의 첨부된 그림과 같이thread1 ->2->3의 순서대로 교차 없이 실행되며 count도 각각 잘 증가한다.



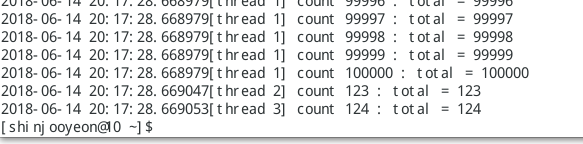


뮤텍스 락을 이용하고 나니 가장 기본적인 형태로 critical section을 보호할 수 있게 되었다. 여러 개의 thread에서 하나의 함수를 실행하는 것이 가능하여 편리하였지만 가능한 변수는 하나인 것이 아쉬운 부분이다. 이를 해결하기 위해선 세마포어 라이브러리를 이용해야 할 것이다.

뮤텍스 관련 코드에선 명백한 결과물과 예측 가능한 결과로 어려움이 많이 없었다 그러나 동기화 작동이 없는 코드에선 수행 시 마다 값이 달라지고 순서가 달라져서 분석에 어려움이 있었다. 따라서 operation 도 계속해서 수정해 보고 순서도 바꿔보고 check변수도 넣어보고 이런 저런 수행을 많이 해 보았다. 그러다 보니 다음과 같은 결과도 나왔다.



위의 사진처럼 thread2와 3이 아예 수행되지 않는 경우도 발생하였다. check변수를 잘 활용하면 이를 줄일 수 있었지만 그래도 실행 중 상당 부분은 위와 같았다. 또한 필자의 컴퓨터에서만 수행시켜봐서 정확한 값은 알 수 없지만 123 번째 count에서 race condition이 가장 많이 발생하였다. 이유는 잘 모르겠지만 노트북의 사양 문제로 추측해본다. 이어서 실행시켜 본 결과도 비슷하다.



thread 가 차례로 alternative하게 수행되어서 count 되어야 하는데, 그렇게 되지 않고, 맨 마지막에 출력되는 thread 2와 3의 경우, total값을 살펴보면 중간에서 수행되어야 할 값이 마지막에 출력되는 오류가 생겨 올바른 counting 값이 아니라는 것을 알 수 있다. 아예 2와 3이 수행 되지 않는 경우도 발생하였다.

마지막으로 조금 아쉬운 부분은 기존의 race condition이 발생하는 예제에서 operation을 바꿔가며 실행하는 부분에서 계속해서 프로그램의 실행이 느려지고 실행 결과가 부정확했다. starvation현상을 줄이려고 시도하면서 여러 가지 exit와 변수를 추가하였는데 동시에 deadlock현상이 발생한 것으로 추측해 본다. 이를 해결하여 구현했었다면 더 결속력있는 코드가 되었을 것이다.

**B. 과제 2**

1. 과제의 개요 및 목적

page fault가 일어나면 운영체제에 트랩이 발생한다. User register와 함께 프로세스의 현 상태를 저장한다. 인터럽트의 원인이 page fault라는 사실을 확인하고 reference가 valid 한지 확인한 다음 디스크에 해당하는 페이지의 위치를 알아낸다.

디스크에 free frame으로의 읽기 요구를 한 다음 그 차례가 오기까지 이 장치의 ready queue에서 기다려야 한다. 디스크에서 찾는 시간과 회전 지연 시간 동안 기다린다.

만약 physical memory에 Free frame이 없다면 victim frame을 선택하여 그 프레임이 dirty면 디스크에 저장한 다음 그 곳에 가져온 페이지를 저장하고, free frame이 있다면 그 곳에 페이지를 저장한다.

기다리는 동안 CPU가 다른 유저에게 할당될 수 있다. 디스크가 다 읽었다고 인터럽트를 건다. 다른 유저의 레지스터와 프로세스 상태를 저장한 다음 인터럽트가 디스크로 온 것을 알아낸다. 새 페이지가 메모리에 올라왔다는 것을 페이지 테이블과 다른 테이블들에 기록한다. CPU가 자기 차례로 오기까지 다시 기다린 다음 저장시켜두었던 레지스터, 프로세스 상태, 새로운 페이지 테이블을 복원시키고 인터럽트 되었던 명령을 다시 실행한다. 어떤 페이지를 교체해야 할 때 기존의 메모리에서 어떤 페이지를 swap out시켜야 하는지에 대한 문제가 생긴다. 가장 대표적인 알고리즘 몇 가지로 이 문제의 답을 찾을 수 있다.

FIFO는 어떤 페이지를 교체해야 할 때 메모리에 올라온 지 가장 오래된 페이지를 내쫓는 알고리즘이다. 페이지가 올라온 시간을 페이지마다 기록해도 되고, 페이지가 올라온 순서로 FIFO Queue를 만들어서 가지고 있어도 된다. 이해가 쉽고 구현도 쉽다. 그러나 성능이 좋지 않다. 잘못된 교체 결정은 page fault rate를 높이고, 프로세스의 실행 속도를 떨어뜨린다. 그리고 일반적으로 프레임의 수가 증가할수록 page fault가 일어나는 수가 감소해야 하는데 FIFO 알고리즘에서는 그 반대 현상이 일어나는 Belady’s Anomaly가 나타난다.

Optimal알고리즘은 앞으로 가장 오랫동안 사용되지 않을 페이지를 찾아 교체하는 알고리즘이다. 프레임 수가 고정된 경우 가장 낮은 page fault rate를 보장한다. 프로세스가 앞으로 메모리를 어떻게 참조할 것인지 예측할 수 없으므로 실제 구현이 어렵다.

LRU는 Optimal algorithm의 근사 알고리즘이다. FIFO 알고리즘이 페이지가 메모리로 들어온 시간을 이용한다면, OPT 알고리즘은 페이지가 앞으로 사용될 시간을 이용한다. LRU 알고리즘은 가장 오랜 기간 동안 사용되지 않은 페이지를 교체하는 것이다. 흔히 Stack을 이용하는데 페이지가 중간에 참조될 때마다 페이지 번호는 스택 중간에서 제거되어 top으로 간다. 여기서 사용하는 스택은 기존의 스택이 아닌 조금 변형된 스택이다. Bottom에 있는 페이지가 victim page이다. OPT만큼은 아니지만 Page fault rate를 감소시킬 수 있다. 그러나 구현하는데 있어서 overhead가 크다.

본 과제에서는 위의 세 가지 알고리즘의 장점과 단점을 살려 알고리즘을 실제로 구현해 보는 것이고, 가상 머신 에서 시뮬레이션 하여 각 알고리즘 별 page fault횟수를 분석하여 실제로 page fault횟수가 증가하고 감소하는지를 알아보려 한다.

2. 프로그램 설계

-FIFO

이 알고리즘의 핵심은 가장 오래 된 페이지가 나가야 한다는 것이다. 처음 든 생각은 이름이 FIFO인 만큼 큐로 구현하면 간편하겠다 라는 생각이 들었다. 또한 들어온 시간을 기록해 두어 이를 보고 변경하는 것도 가능하겠다고 생각했다. 그렇게 고민하다 문득 어차피 하나씩의 페이지가 들어오고 하나씩의 페이지가 나가니 가장 먼저 들어온 페이지 하나씩만의 위치를 저장해 두면 되겠다는 생각을 하게 되었다. 그래서 본 알고리즘의 구현은 복잡하지 않고 굉장히 간단하게 먼저 들어온 것을 저장하는 변수를 하나 정해두는 것으로 구현 설계하였다.

-OPT

이 알고리즘의 핵심은 앞으로 가장 나중에 사용될 페이지를 찾는 것이다. 이러한 구조의 구현은 지금 frame안의 index와 string reference의 index의 차이를 배열에 저장하는 형식으로 구현을 생각하였다. 이 차이 값을 기준으로 하여 언제 frame안의 page number가 나오게 될지를 같은 index의 다른 배열에 가지고 있는다.

-LRU(ref count)

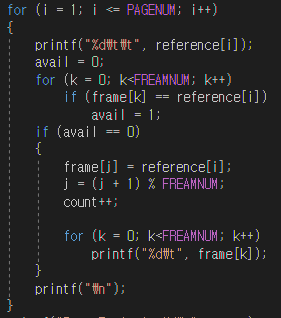
가장 최근에 사용된 시간을 저장하는 변수를 정해둔다. 만약 참조가 일어날 경우 이 값을 업데이트 하며 page fault가 되는 경우는 이 변수의 값이 가장 이전의 값인 경우일 때로 한다. 한번 수행될 때 마다 참조되지 않으면 +1시켜준다 따라서 각 프레임마다 counter변수가 필요하다. 이 값이 가장 큰 것을 내보내면 된다.

-LRU(second chance)

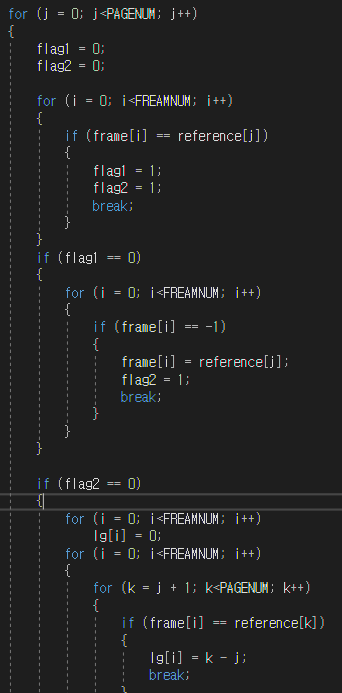
마찬가지고 LRU알고리즘인데 여기선ref bit를 이용하여 이것이 1인 것을 찾아 없애고 새로운 page를 넣은 후 다시 0으로 바꿔준다. 하나씩 돌아가면서 값을 검사하고, 1인 것은 찾아 내보내면서 새로 들어온 것을 0으로 바꿔주는 작업을 반복한다. 언제 ref되는지 세세하게 보지 않아도 된다.

3. 프로그램의 주요부분 설명

-FIFO

이 for문 안에서 사실상 모든 작업이 이루어진다. for문을 순서대로 돌면서 ref string과 값이 다를 경우count를 증가시켜 구현하는 코드이다. 두 번째 for문은 참조 값이랑 같은 경우 이를 확인하기 위해 avail변수를 1로 만들어 주는 부분이다. 이는 현재 frame에 들어있는 page number중에서 ref string과 같은 것이 있는 경우를 말한다.

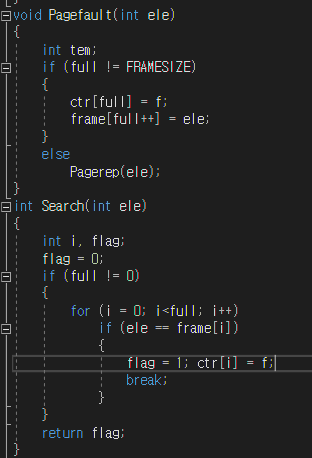
-OPT



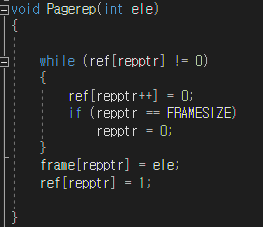
여기서 가장 핵심적인 부분은 flag로 상황을 구별한 코드이다. 현재 프레임 안의 page num과 ref string의 값이 같으면 이 flag 1, 2의 값이 1이 되어 break한다. 다르면 탐색을 진행한다. 탐색은 flag2 if문 안의 코드를 말한다. 이 안에서는 같은 값이 올 경우 lg 배열의 값을 1로 하고 frame 값과 ref string 값 과의 차이를 구하여 0 인 경우 lg 배열의 값이 0 인 if문) 바로 온다는 것을 나타내어 index에 i를 저장하고 found를 1로 바꾼다. 아닐 경우 큰 값을 찾기 위한 max변수 이용 for문을 진행한다.

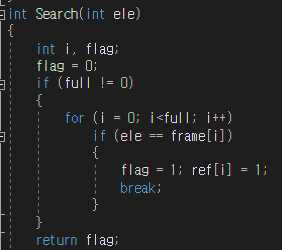
-LRU(ref count)

마찬가지고 LRU알고리즘인데 여기선ref bit를 이용하여 이것이 1인 것을 찾아 없애고 새로운 page를 넣은 후 다시 0으로 바꿔준다. 하나씩 돌아가면서 값을 검사하고, 1인 것은 찾아 내보내면서 새로 들어온 것을 0으로 바꿔주는 작업을 반복한다. 언제 ref되는지 세세하게 보지 않아도 된다.

이처럼 f에 현재 시간을 저장해서 모든 함수에서 사용하고 ctr배열을 각 frame마다 가진 카운터로 사용하여 index로 연결하였다. else문에 있는 함수에서 ctr값을 비교하여 가장 오래 전에 참조된 값을 내보낸다.

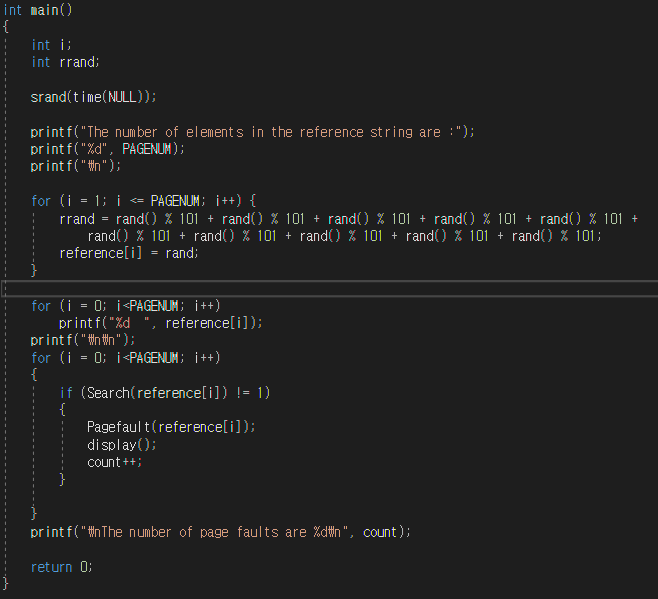
--LRU(second chance)



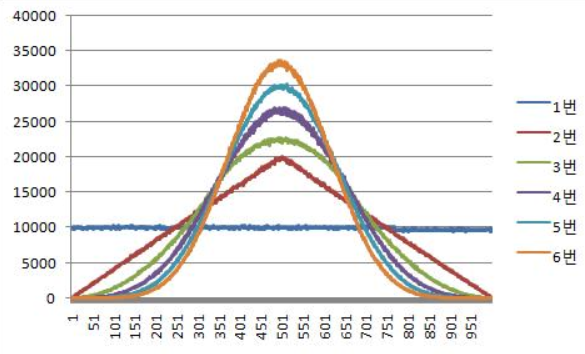


ref bit를 사용하여 탐색이 지난 경우 ref값을 1로 만들어 주고 내보낼 페이지를 결정 할 때 0이 아닌 페이지를 while문 돌려서 검사한다. 또 다시 검사가 끝난 페이지는 1로 바꿔준다.

4. 프로그램 실행조건 및 상황설명

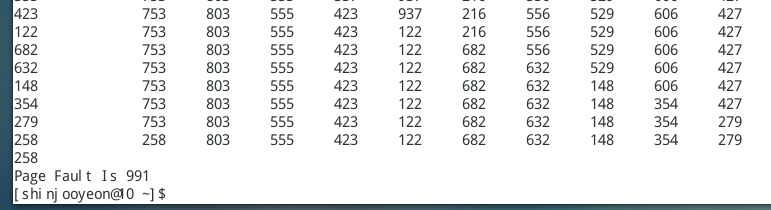


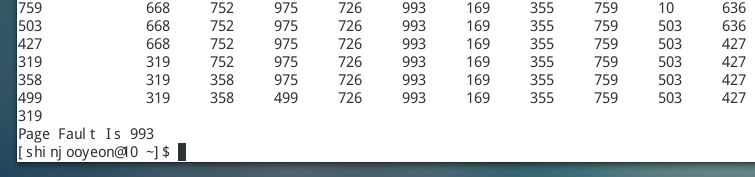
위 사진은 second chance알고리즘의 메인이다. 정규 분포를 따르는 난수 생성을 어떻게 구현할 지 많이 고민해 보았고 실제로 math.h 함수를 이용한 정규 분포 함수 구현은 완료하였었다. 정규 분포 함수를 먼저 구현한 후 (-1 부터 1까지의 값이 종 모양으로 생성됨) 이에 절대값을 취한 후 반올림 은 0.5를 더하여 floor를 취하여 만들고 이를 원하는 크기로 scale시키는 형태로 구현하였다. 그러나 리눅스에서 sqrt와 같은 math.h함수가 동작하지 않았고 새로운 방법을 고민하던 중 난수를 더하면 더할수록 정규분포에 가까워 진다는 것을 알게 되었다.



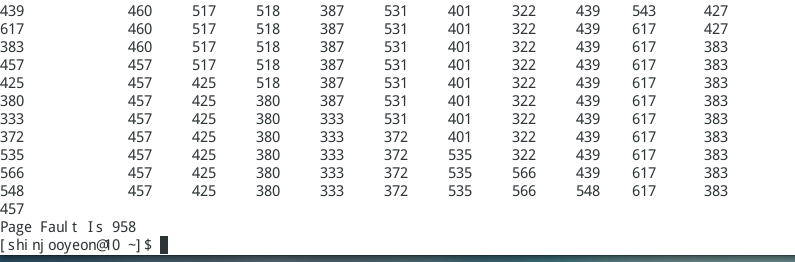
아래 그래프는 난수를 더하면 더할수록 정규분포에 가까워짐을 테스트하여 증명한 그래프이다. 따라서 이처럼 0-1000까지의 난수는 0-100까지의 난수를 10번 생성하여 더하는 것으로 구현하였으며 실제로 결과값을 비교해보니 정규분포를 따를 때 처럼 page fault가 감소하는 것을 눈으로 확인하였다.

5. 프로그램 실행결과 스크린샷 및 설명

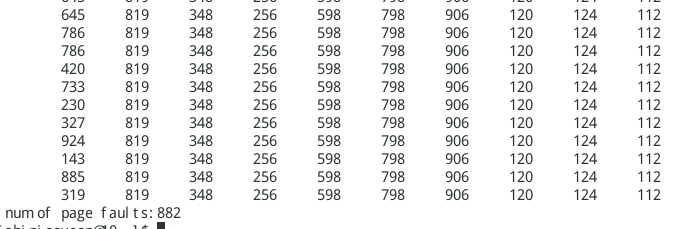




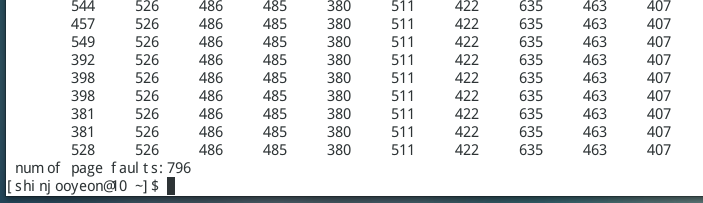
-정규분포를 따르지 않는 FIFO-> 상당히 큰 page fault값을 가진다.



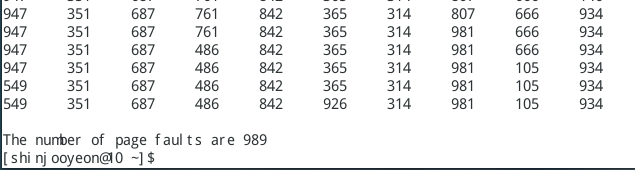
-정규분포를 따르는 FIFO-> 결과값 중 작은 값은 958까지도 출력되었다. 보통은 이보다 크다.



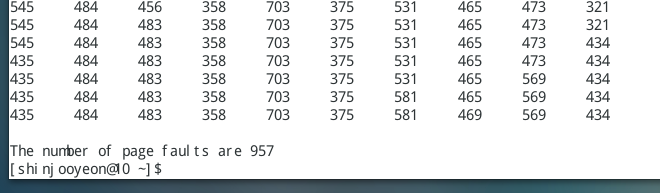
-정규분포를 따르지 않는 OPT->정규분포 없이도 상당히 작은 값이다



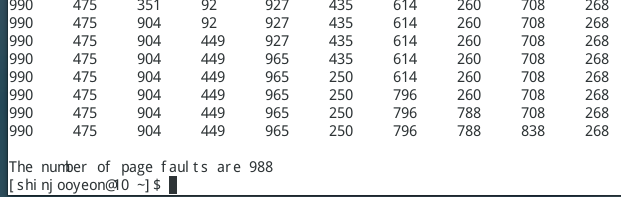
-정규분포를 따르는 OPT->정규 분포를 따르고 나니 확연히 작아진다.



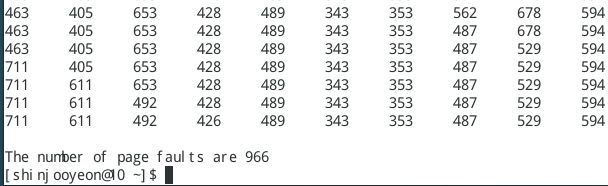
-정규분포를 따르지 않는 ref ->거의 FIFO와 비슷하지만 그래도 조금 더 작은 값이다. OPT보다는 상당히 크다.



-정규분포를 따르는 ref ->상당히 줄어들었지만 전체적인 구조는 FIFO와 비슷하다.

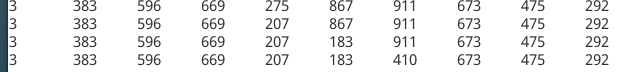


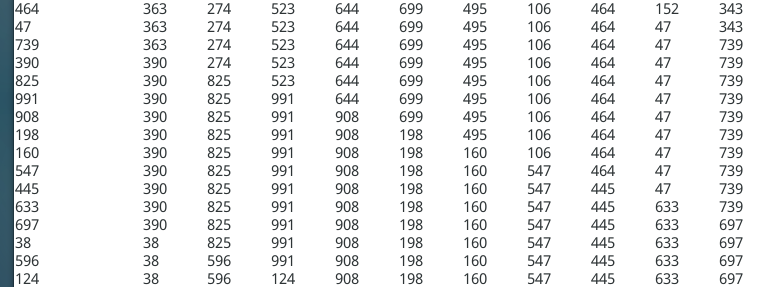
-정규분포를 따르지 않는 sec->같은 알고리즘인 만큼 ref와 비슷하다.



-정규분포를 따르는 sec-> 역시 ref와 비슷한 결과를 보인다. 이전의LRU보다 높은 값이다.

정규 분포를 따르지 않는 경우 밑의 사진처럼 숫자의 편차가 몹시 커서 상당한 차이를 보였다.





<결과 분석>

각각 의 경우=정규아님/정규 로 나타낼 때

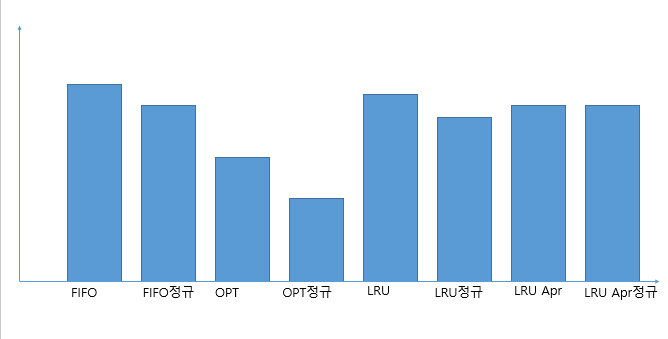
FIFO = 986 991/958 993

OPT =882 884 /793 796

REF = 985 989/957 958

SEC = 986 988/971 976

위의 결과와 같았고 이는 그래프로 나타내면 위의 순서대로 밑의 그래프와 같다.



두 개씩 묶어서 볼 경우 확연히 optimal은상당히 낮은 page fault를 보이고 FIFO가 가장 높다. 먼저 가장 빠르게 구현 가능하며 구현이 쉽고 코드도 짧고 간단한 알고리즘은 FIFO이다. 그러나 성능이 좋지 않다. 잘못된 교체 결정을 하여page fault rate를 높이고, 프로세스의 실행 속도를 떨어뜨린다. 그리고 일반적으로 프레임의 수가 증가할수록 page fault가 일어나는 수가 감소해야 하는데 FIFO 알고리즘에서는 그 반대 현상이 일어나는 Belady’s Anomaly가 나타난다. 따라서 공간적으로 비효율적임이 증명된다. 프레임의 수를 늘려도 소용이 없는 격이기 때문이다. 다음으로 OPT는 굉장한 비용적 효율을 보여준다. 조금도 아닌 확연한 차이로 적은 수의 page fault를 보여주어 놀랐다. 그러나 프로세스가 앞으로 메모리를 어떻게 참조할 것인지 예측할 수 없으므로 실제 구현이 어렵다. 또한 코드도 굉장히 많은 반복문을 거쳐서 상당히 복잡하고 긴 코드로 구현되었다. 이어서 LRU의 경우 OPT보다는 아니지만 Page fault rate를 감소시킬 수 있었다. 이는 현실적으로 실현 불가능한 OPT와 구현은 쉽지만 효율이 떨어지는 FIFO를 생각해 보면 가장 현실적이고 실제로 가장 많이 사용되는 알고리즘이다. 그러나 구현 코드가 OPT보다도 길고 사실상 결과물이 FIFO와 크게 다르지 않은 것으로 나타난 것을 보아 구현하는데 있어서 overhead가 크다는 것을 알 수 있다. 또한 구현 시 변수도 상당히 많이 들어서 메모리 소요도 많았을 것으로 예상된다. 마지막으로 LRU-approx Algorithm의 경우 LRU 보다는 아니지만 Page fault rate를 감소시킬 수 있다. 그러나 최근에 조금 썼던 페이지는 남고 과거에 많이 썼었지만 최근에 쓰지 않은 페이지가 교체될 수 있다는 단점이 있다. 이는 메모리 공간의 효율에서 상당히 좋지 않은 부분이다.