REPORT

제목: lab2_sync

과목명 : 운영체제
담당교수: 최종무 교수님
이름: 최지윤
학번: 32194747
제출일: 2020.

단국대학교
Dankook University

I. 문제 정의

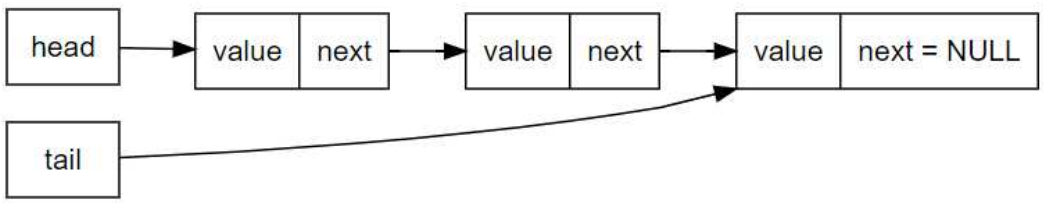
자동차 생산자-소비자 문제

> OO공장에서 5종류의 차량을 생산한다. OO공장에는 차량 생산 라인은 하나만 존재한다. 차량 생산자는 차량 생산 라인에 차량을 Round Robin 방식으로 출고를 진행한다. 이에 5명의 구매자들은 생산 라인에 출고된 차량을 앞에서 부터 하나씩 가져갈 수 있다. 생산 라인에 출고된 차량이 없는 경우 구매자는 차량을 가져갈 수 없다. 생산자는 생산 라인이 가득 찬 경우 새롭게 차량을 출고할 수 없다. 현재 상황을 가정해 코드를 구현해라. (Time quantum = 1, 4)

이 문제의 핵심은 queue 자료구조, RR 스케줄링 방식, Synchronization, Lock 등의 구현이다. 다음은 과제를 이해하기 위한 이론적인 내용이다.

- Concurrent Queue

Queue 자료구조는 기본적으로 FIFO (First-In-First-Out) 데이터 구조를 제공한다. Concurrent Queue(동시 큐)는 데이터 구조를 명시적으로 잠글 필요 없이 큐에 읽고 쓰는 여러 스레드를 허용하도록 설계되었다. 아래 그림과 같이 Concurrent Queue는 Node가 연결되어 있는 구조이며 첫 번째 노드는 head가, 맨 마지막 노드는 tail가 가르키고 있다. 이때, enqueue를 하기 위해선 현재 tail이 가리키고 있는 노드 뒤에 새로운 노드를 연결하고 tail을 옮겨준다. dequeue를 하기 위해서는 head를 다음 노드로 바꾸어주면 된다. Node로 표현되기 때문에 count(노드 수)의 최대 크기는 정해져 있지 않지만 count가 음수가 될 순 없음을 유의해야한다.



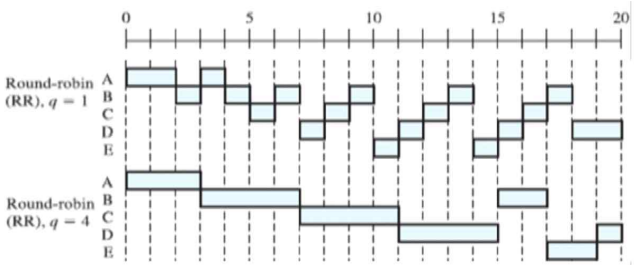
- RR Scheduling

Round Robin은 Time-sharing 시스템을 위해 설계된 선점형 스케줄링 방식이다. Process 끼리 우선 순위를 두지 않고, 순서대로 Time quantum에 따라 CPU로 할당하는 방식이다. 구체적으로, 가장 먼저 들어온 process가 할당 받은 시간(Time quantum)동안에만 실행 후 다음 process가 시간을 할당 받는다. 각 process는 ready queue에서 순서를 기다린다. RR 스케줄링의 예시는 다음과 같다.

✔ Workload : 5 tasks

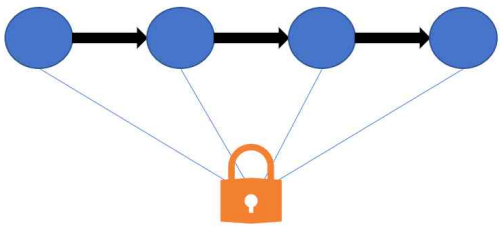
Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

✔ Scheduling Result

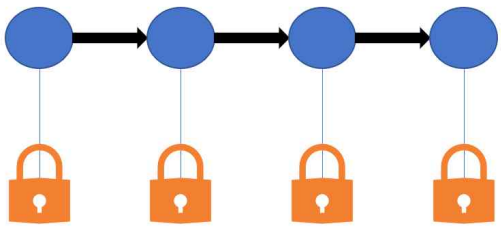


- Coarse-grained Lock & Fine-grained Lock

Multi-thread 등의 임계영역이 존재하는 환경에서 경쟁상태를 방지하기 위해 Lock 을 사용할 수 있다. 하지만 경쟁상태가 발생하지 않더라도 Lock 을 사용하는 방식에 따라 그 성능이 크게 좌우될 수 있다. Lock 을 사용하는 방식은 임계영역의 선정 범위에 따라 즉 Lock 을 얼마나 세분하게 설정하냐에 따라 Coarse-grained Lock, Fine-grained Lock 의 두 가지 방식으로 사용될 수 있다. Coarse-grained Lock 방식은 임계영역의 범위를 크게 잡아 많은 양의 데이터를 한번에 보호하는 방식이며, Fine-grained Lock 은 임계영역의 범위를 세분화하여 각 부분마다 Lock 을 사용하는 방식이다.



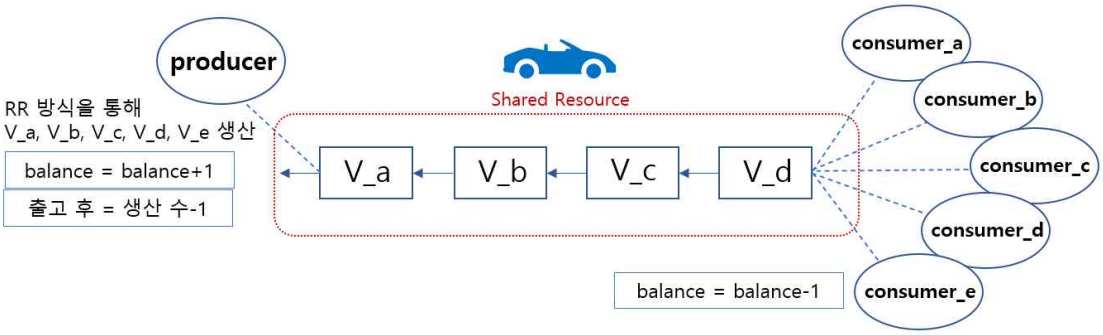
Coarse-grained Lock



Fine-grained Lock

다음은 자동차 생산자-소비자 문제를 표현한 이미지이다.

Producer thread 는 설정된 총 생산 수, time quantum, workload 에 따라 5 개의 차량들을 RR 로 스케줄링하여 공유 자원인 생산 라인(Queue)에 넣어준다. 5 개의 Consumer thread 는 생산 라인에 출고되어있는 차량을 순서대로 구매하여 Queue 에서 빼준다.



II. 설계 및 구현

- 생산 Workload

	생산 시작 시간	생산 수
V_a (1)	0	Total / 5
V_b (2)	2	Total / 5
V_c (3)	4	Total / 5
V_d (4)	6	Total / 5
V_e (5)	8	Total / 5
Total		100 ~ 10000

Producer가 차량을 생산하기 위한 workload는 위와 같다.
각 차량의 생산 수는 총 생산 수를 5로 나눈 값으로 하여 일정하게 분배하였다.

- Concurrent Queue

차량 공유 자원을 Concurrnt Queue로 관리하기 위해서 car_queue(CQ)라는 구조를 만들었다. 생산자가 큐에 접근하여 차량을 추가할 때는 새로운 노드를 만들어 큐의 마지막(rear)에 연결해주고 rear를 새로운 노드로 재지정 해준다. 이와 같은 과정은 원자적으로 수행되어야하므로 노드를 연결하고 rear 재지정하는 부분은 임계 영역이다. 소비자가 큐에 접근하여 차량을 빼낼 때는 기존의 head가 가리키고 있던 노드를 새롭게 head로 지정한 후 기존 head에 해당하던 노드는 free 시켜주어 구현했다.

- RR Scheduling

RR 스케줄링은 이미 lab1_sched 과제에서 구현한 바 있다. 이 코드를 그대로 사용하였다. 우선 RR 스케줄링을 위한 scheduling_queue(SQ)를 만들고, 생산을 시작한 차량이 있으면 enqueue, 입력받은 time quantum 만큼 생산을 완료하면 dequeue, 아직 생산 수량이 남았다면 다시 enqueue 하는 RR의 원리를 그대로 적용하여 구현했다. 이렇게 workload에 따라 5대의 차량 스케줄링을 먼저 시켜 Print라는 배열에 차량 번호를 넣어주었다. 이후 Print 배열에 순서대로 들어가 있는 차량을 car_queue에 put하여 생산 라인에 차량을 출고시키는 시스템이다. 구현 후 차량이 출고되는 시점마다 어떤 차량인지 출력해보는 printf문을 추가하여 보았더니 실제로 RR 스케줄링 방식에 따라 제대로 수행되었음을 확인하였다.

- (Coarse-grained Lock || Fine-grained Lock) && Conditional variable

운영체제 7주차 강의에 Semaphore를 사용하여 Lock과 conditional vairiable을 표현하는 내용을 배웠는데 과제 구현을 7주차 이전부터 시작한 바람에 간단한 semaphore 대신 pthread_mutex_t와 pthread_cond_t를 사용했다.

자동차 생산 라인에 해당하는 car_queue 는 공유 자원이기 때문에 두 개 이상의 스레드가 동시에 접근하면 문제가 발생할 수 있다. 따라서 mutex lock 을 통해 스레드의 상호 배제를 보장해 주어야 한다. Coarse-grained Lock 으로 구현할 때는 p_mutex 라는 하나의 락 변수를 사용하여 car_queue 노드를 한꺼번에 관리했다. Fine-grained Lock 으로 구현할 때에는 rearLock 과 frontLock 이렇게 두 개의 락 변수를 사용했다. 즉, 각각 다른 락을 사용하여 put 과 get 상황에서 car_queue 의 노드들을 보호(?)하고, 병행성을 더욱 높여주었다.

보통의 생산자 소비자 문제에서는 버퍼가 꽉 찼을 때, 생산자는 wait 해야하는 로직이 있는데 이 문제 상황에선 생산 Queue 에 들어갈 수 있는 차량의 최대 수는 없다고 했으므로 따로 조건 변수를 설정하여 cond_wait 을 시켜줄 필요가 없었다. 하지만 차량 count(balance)는 음수가 될 수는 없으므로 생산 Queue 가 비었을 때 소비자에 대해서는 제어를 해주어야 한다. 따라서 조건 변수 fill 을 선언해서 Queue 의 balance 가 0 이면 소비자는 Queue 에 접근하지 못하고 대기하도록 설계하였다.

최종적으로 자동차 생산자-소비자 문제 구현에 사용된 function 의 종류는 다음과 같다.

```
void init_car_queue(CQ* q);              // 생산 Queue 초기화
void init_scheduling_queue(SQ* q);       // 스케줄링 Queue 초기화
void enqueue(SQ* q, int c_n);            // RR 스케줄링을 위한 enqueue
Node* dequeue(SQ* q);                    // RR 스케줄링을 위한 dequeue
void RR();                               // RR 스케줄링
void put_car(CQ* q, int c_n);            // 생산 Queue에 put (enqueue)
void get_car(CQ* q);                     // 생산 Queue에서 get (dequeue)
void* producer_n(void* qq);              // No Lock으로 구현한 producer
void* producer_c(void* qq);              // Coarse-grained Lock으로 구현한 producer
void* producer_f(void* qq);              // Fine-grained Lock으로 구현한 producer
void* consumer_n(void* qq);              // No Lock으로 구현한 consumer
void* consumer_c(void* qq);              // Coarse-grained Lock으로 구현한 consumer
void* consumer_f(void* qq);              // Fine-grained Lock으로 구현한 consumer
void show(int balance, double result);   // 결과 출력
```

III. 수행 결과 및 분석

- Total_car (c) = 100, Time_quantum (q) = 1

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=100 -q=1
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000261
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000580
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000183
```

- Total_car (c) = 1000, Time_quantum (q) = 1

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=1000 -q=1
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.000311
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.003906
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.002149
```

- Total_car (c) = 1000, Time_quantum (q) = 1

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=10000 -q=1
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.001027
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.063452
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.027203
```

- Total_car (c) = 100, Time_quantum (q) = 4

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=100 -q=4
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000171
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000373
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000109
```

- Total_car (c) = 1000, Time_quantum (q) = 4

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=1000 -q=4
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.000251
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.004405
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 1000
        Final Balance Value = 0
        Execution time = 0.002190
```

- Total_car (c) = 1000, Time_quantum (q) = 4

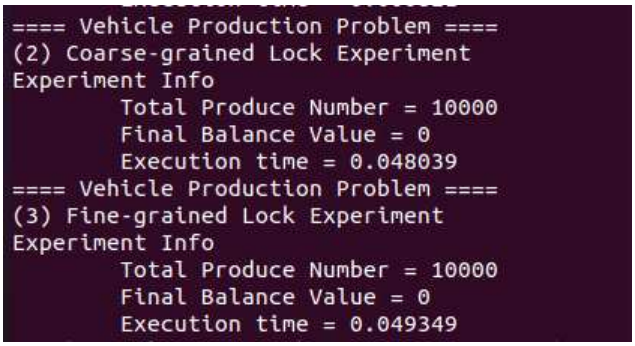
```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ sudo ./lab2_sync -c=10000 -q=4
==== Vehicle Production Problem ====
(1) No Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.000850
==== Vehicle Production Problem ====
(2) Coarse-grained Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.061223
==== Vehicle Production Problem ====
(3) Fine-grained Lock Experiment
Experiment Info
        Total Produce Number = 10000
        Final Balance Value = 0
        Execution time = 0.001305
```

(1) Lock의 유무에 따른 차이

컴퓨터 동작 상의 특성인지, 코드 흐름 상 상호배제의 원리가 내재되어 있는지는 잘 모르겠지만 실습 상황에서는 Lock을 걸어주지 않아도 Final balance value가 0이 나와 모든 수량의 차량들이 생산되었다가 소비되었음을 확인하였다. 하지만 이는 굉장히 운이 좋은(?) 케이스라고 보아야 한다. Lock을 통해 스케줄링을 제어해주지 않으면 공유 자원에 대해 스레드 간의 경쟁 상태가 심각해지면서, 경우에 따라 잘못된 값이나 누락된 결과가 나타날 수 있다. 따라서 여러 스레드가 공유하고 있는 생산 Queue에 대해 Lock을 걸어 상호배제를 보장해주고, 스레드 간의 스케줄링을 제어해주어야 한다.

(2) Coarse-grained / Fine-grained Lock에 따른 차이

Coarse-grained Lock은 모든 임계 영역(생산 Queue 관련)을 큰 단위 즉, 하나의 Lock으로 관리한 것이다. 반면, Fine-grained Lock은 임계영역을 작은 단위 즉, 여러 개의 Lock을 선언하여 관리했다. 확실히 사용하는 Lock의 개수가 이전보다 늘어나기 때문에 구현하는 데에 있어서 Coarse-grained Lock보다 까다로운 부분이 있었다. 하지만 실행 결과에서 보면 알 수 있듯 전체 차량 수와 time_quantum과는 상관없이 Fine-grained Lock으로 구현했을 때가 Coarse-grained Lock으로 구현했을 때보다 수행시간이 짧게 관측되었다. 물론 예외적인 상황도 있었다.



위의 결과에서는 Fine-grained Lock Experiment가 Coarse-grained Lock보다 Execute time이 짧다. 이는 특정한 경우에 따라 Fine-grained Lock에서의 락 경쟁이 치열해져 나타난 결과라고 보여지며, 대부분의 경우 Fine-grained Lock에서의 Execute time이 더 짧게 나타났다. 차량을 생산하고 소비한다는 같은 작업을 하는데도 불구하고 수행시간이 더 짧다는 것은 그만큼 성능이 더 좋다는 것을 의미한다. 따라서 Fine-grained Lock으로 구현하면 과정은 조금 힘들지만 병행성을 보장해주어 성능을 더 높일 있다는 장점을 확인할 수 있었다.

(3) 전체 생산 차량 수에 따른 차이

전체 생산 차량 수(Total_car)를 100, 1000, 10000으로 늘려가며 수행해 보았다. 예상했던 대로 전체 차량 수가 클수록 수행시간이 더 오래 걸리는 것을 확인했다. 그만큼 반복문을 많이 돌기도 하고, 락 경쟁이 발생하는 빈도도 많을 것이라고 추측할 수 있다. 식당에서 손님이 많을수록 음식 나오는 속도가 느려지는 것과 비슷하다.

IV. 논의

이번 과제는 많이 어렵긴 했지만 과제 내용 자체가 신선했다. 수업 시간에 흥미롭게 들었던 생산자-소비자 문제에 자동차 생산이라는 스토리가 더해져서 새롭게 느껴졌다. 처음에는 문제에 살짝은 생소한 '자동차 생산라인', '출고'라는 단어들이 섞여있어 이해하기 어려울 것이라 생각했지만 자세히 읽어보니 그저 다중 스레드를 생성하고, 공유 자원과 임계 영역에 대해 어떻게 상호배제를 보장하며 병렬성을 높일 수 있을지에 대한 간단한 문제였다. 문제 자체는 간단해 보이지만 구현 과정은 절대 그렇지 않았다. 다양한 시행착오를 예상하여 lab2과제가 나오자마자 접근해보았지만 segmentation fault의 늪에서 빠져 나오는데에 2주가 훨씬 넘는 시간이 걸렸다.

segmentation fault는 대부분의 경우 디버깅을 해도, 구글링을 해도 원인을 쉽게 찾을 수 없었다. 나는 원인을 찾기 위해 코드 사이사이 로직이 생각대로 잘 돌아가는지 확인하기 위한 print문을 넣어 일일이 어떻게 스케줄링이되고, 어떤 값들이 생산 Queue에 들어가는지 지켜봤다. 그렇게 겨우 찾아낸 첫 번째 원인은 RR 스케줄링을 구현하는 부분에 있었다. 정확한 이유는 모르겠지만 스케줄링을 위한 반복문을 total_car만큼 수행하는 부분과 enqueue 및 dequeue하는 부분이 서로 맞지 않아 반복문이 생각보다 많이 돌게되어 문제가 생긴 듯했다. 그래서 기존에 열심히 구현했던 RR부분은 모두 지우고 lab1 과제에서 구현했던 RR 코드를 그대로 가지고 와서, 순서만 맞추어 생산자가 생산 Queue에 차량을 put하는 방식을 선택했다. 원래는 RR 스케줄링 로직을 버리려고 했는데 그동안 해온 노력이 아까워서 조금 더 붙잡아 보다가 어느 순간 segmentation fault가 발생하지 않는 실행 결과를 보고 기쁨을 감출 수 없었다. 이때까지는 과제가 성공적으로 끝날 줄 알았다. 하지만 곧 또다시 segmentation fault의 늪에 빠져버렸다. 바로 No Lock, Coarse-grained Lock, Fine-grained Lock 구현을 위해 함수를 분리했을 때였다. 처음에 Coarse-grained Lock으로 구현을 완료하여 정상 수행됨을 확인하고서 함수를 나눠 하나는 Lock을 지우고, 하나는 두 개의 락을 사용하는 방향으로 코드를 수정해보았더니 segmentation fault가 발생했다. 원인은 알 수 없었지만 우선 각 함수에서의 스레드 연관성을 완전히 차단시키기 위해 생산 Queue와 pthread를 모두 분리시켰다. 그래도 해결이 되지 않아 확인해보니 Coarse-grained Lock과 Fine-grained Lock 환경에서는 전혀 문제가 없는데 Lock을 완전 제거하였을 때 문제가 있음을 알 수 있었다. 정확히는 생산 Queue가 비었을 때 소비자 스레드가 스케줄링 되었을 때 문제가 있는 것이었다. 이는 put_car 함수에서 생산 Queue의 balance가 0이면 return 0을 해주는 방식으로 해결했다.

과제를 마무리하면서 여러 고난을 겪으며 무엇을 배웠는지 생각해보았다. 가장 크게 배운 것은 Coarse-grained와 Fine-grained Lock이 각각 어떻게 상호배제를 보

장하고. 어떻게 다르게 동작하는지, 멀티 스레드 환경에서 Lock을 써주지 않으면 어떤 문제가 발생하는 지 등인 것 같다. 다음에 기회가 된다면 Fine-grained Lock 방식으로 구현했을 때 Lock의 개수에 따라 Lock Overhead가 얼마나 발생하고, 성능에는 어떻게 영향을 미치는 지에 대한 내용을 실습해보고 싶다.