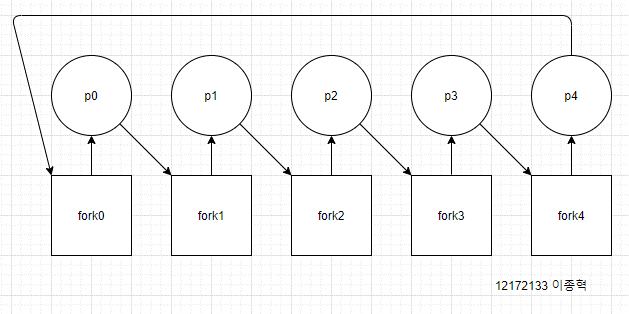
오퍼레이팅 시스템

Project 3 – Dining Philosopher Problem

12172133 이종혁

1. 식사하는 철학자 문제 자원 할당 그래프

* 아래 그림은 식사하는 철학자 문제의 process와 resource 관계를 나타내는 자원 할당 그래프이다.



* 각각의 p들은 철학자의 역할로, 프로세스가 되어 각각의 fork 자원을 요구할 것이다.
* 각각의 철학자의 왼쪽에 있는 식기는 철학자의 번호와 같으며, 각 철학자의 우측에 있는 식기는 p의 n+1 fork라고 가정한다.
* 각 철학자들은 왼쪽의 포크부터 사용하므로, 동일 번호의 식기를 점유하고 우측의 포크인 n+1번 식기를 요구하게 된다.
* n+1번 식기를 요구했으나 이미 그 식기가 점유한 상태라면, 철학자는 그 식기를 얻기 전까지는 식사도 생각도 하지 않는다.
* 각 프로세스가 자원을 점유한 상태에서 다른 프로세스가 점유한 자원을 요구하므로, 식사와 생각을 하기 이전에 다른 데이터가 점유한 자원을 받기 위해 계속해서 기다리게 되어 Deadlock 현상이 나타날 수 있다.

1. 식사하는 철학자 문제 해결
2. No hold and wait

* hold and wait은 이미 자원을 보유하고 있는 프로세스가 다른 프로세스의 자원을 요청하면서 기다리고 있는 상태이므로, 이를 없애기 위해 프로세스에 자원을 할당할 때 필요한 모든 조건을 한번에 할당하는 방식을 이용한다.
* 아래는 문제 해결을 위한 소스코드이다.

**package** Dining\_philosopher;

**import** java.util.concurrent.Semaphore;

**class** Philosopher1 **extends** Thread {

**int** id;

Semaphore lfork, rfork, once;

Philosopher1(**int** id, Semaphore lfork, Semaphore rfork, Semaphore once) {

**this**.id = id;

**this**.lfork = lfork;

**this**.rfork = rfork;

**this**.once = once;

}

**public** **void** run() {

**try** {

**for**(**int** i = 0; i<100; i++) {

once.acquire();

lfork.acquire();

rfork.acquire();

once.release();

eating();

lfork.release();

rfork.release();

thinking();

}

}**catch** (InterruptedException e) { }

}

**void** eating() {

System.***out***.println("[" + id + "] eating");

}

**void** thinking() {

System.***out***.println("[" + id + "] thinking");

}

}

**public** **class** philo1 {

**static** **final** **int** ***num*** = 5;

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**int** i;

Semaphore[] fork = **new** Semaphore[***num***];

**for** (i=0; i<***num***; i++)

fork[i] = **new** Semaphore(1);

Semaphore once = **new** Semaphore(1);

Philosopher1[] phil = **new** Philosopher1[***num***];

**for** (i=0; i<***num***; i++)

phil[i] = **new** Philosopher1(i, fork[i], fork[(i+1)%***num***], once);

**long** sx = System.*currentTimeMillis*();

**for** (i=0; i<***num***; i++)

{

phil[i].start();

}

**for**(i = 0; i<***num***; i++)

phil[i].join();

**long** ex = System.*currentTimeMillis*();

System.***out***.println( (ex - sx) +"ms");

}

}

텍스트이(가) 표시된 사진

자동 생성된 설명

* 한번에 모든 자원을 할당하기 위하여, lock을 걸어줄 세마포어 변수 once를 추가한다.
* 스레드의 run함수에서 세마포어 once로 lock을 걸어둔 채 철학자의 좌 우측의 포크를 모두 할당시킨다면, 이 스레드는 필요한 자원을 모두 받았으므로 wait 상태가 되지 않는다. 즉 언젠가 식기 두개를 모두 release할 것이므로 다른 철학자들의 wait상태를 해소해 줄 수 있는 것이다.

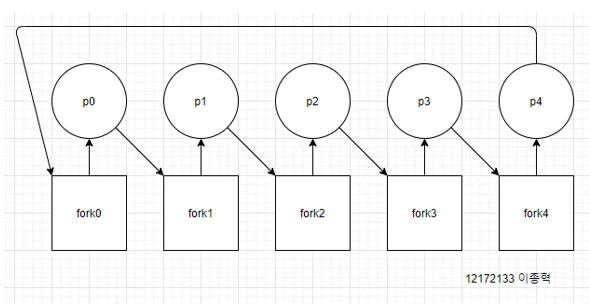
텍스트이(가) 표시된 사진

자동 생성된 설명

* once는 lock & unlock을 위한 세마포어 변수이다. 따라서 Semaphore을 1로 초기화해주어 스레드가 공유할 수 있는 자원을 1개로 제한한다.
* run 함수에서 각 스레드가 once.acquire()을 호출하였을 때, 자원이 1일 때 호출한 스레드만이 run 함수를 계속 진행할 수 있게 된다. 자원이 1보다 작을 때 호출한 스레드들은 모두 wait상태가 된다.

1. Removing Circular Wait 1

* 아래 그림과 같은 문제 상황에서, 모든 철학자가 왼쪽의 포크를 집어버린 상태라면 Deadlock이 걸리게 된다. 즉 closed chain이 생겨 환형이 만들어지게 되는 것이다.



* 이 문제를 해결하기 위해서는 특정 철학자가 식기를 집는 order에 변화를 주어 chain을 끊어주어야 한다. 그 소스코드는 아래와 같다.

**package** Dining\_philosopher;

**import** java.util.concurrent.Semaphore;

**class** Philosopher2 **extends** Thread {

**int** id;

Semaphore lfork, rfork;

Philosopher2(**int** id, Semaphore lfork, Semaphore rfork) {

**this**.id = id;

**this**.lfork = lfork;

**this**.rfork = rfork;

}

**public** **void** run() {

**try** {

**for**(**int** i = 0; i<100; i++) {

**if**(id < 4) {

lfork.acquire();

rfork.acquire();

}**else** {

rfork.acquire(); // r0

lfork.acquire(); // r4

}

eating();

lfork.release();

rfork.release();

thinking();

}

}**catch** (InterruptedException e) { }

}

**void** eating() {

System.***out***.println("[" + id + "] eating");

}

**void** thinking() {

System.***out***.println("[" + id + "] thinking");

}

}

**public** **class** philo2 {

**static** **final** **int** ***num*** = 5;

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**int** i;

Semaphore[] fork = **new** Semaphore[***num***];

**for** (i=0; i<***num***; i++)

fork[i] = **new** Semaphore(1);

Philosopher2[] phil = **new** Philosopher2[***num***];

**for** (i=0; i<***num***; i++)

phil[i] = **new** Philosopher2(i, fork[i], fork[(i+1)%***num***]);

**long** sx = System.*currentTimeMillis*();

**for** (i=0; i<***num***; i++)

{

phil[i].start();

}

**for**(i = 0; i<***num***; i++)

phil[i].join();

**long** ex = System.*currentTimeMillis*();

System.***out***.println( (ex - sx) +"ms");

}

}

텍스트이(가) 표시된 사진

자동 생성된 설명

* run함수에서 id가 0~3인 철학자들은 기존 포크를 드는 방식을 고수한다.
* 마지막 자리에 위치한 철학자는 포크를 드는 순서를 반대로 한다.
* 만약 0~3인 철학자가 왼쪽의 포크만 든 채로 4번 철학자 쓰레드로 넘어왔다고 할 때, 4번 철학자는 0번 철학자 기준으로 왼쪽의 포크를 먼저 요구하게 된다. 그러나 0번 철학자는 왼쪽 포크를 이미 점유한 상태이므로, 4번 철학자는 아무 자원도 점유하지 못한 채 wait상태로 간다. 따라서 환형 대기를 해결할 수 있게 된다.

1. Removing Circular Wait 2

* ㄴ에서 이미 설명한 Deadlock 상황을 해결하기 위해, 각 철학자들의 식기를 집는 규칙을 정해준다.
* 짝수 번호를 가진 철학자는 왼쪽 포크를 먼저 집고, 홀수 번호를 가진 철학자는 우측 포크를 먼저 집도록 한다.
* 이렇게 되면 어느 철학자가 바로 옆의 철학자 쪽의 식기를 요구할 때, 이미 점유하고 있는 식기를 먼저 요구하게 되므로 결국 자원을 할당받지 못하고 wait되게 된다. 환형 대기를 해결할 수 있다.
* 이와 같은 코드는 아래와 같다.

**package** Dining\_philosopher;

**import** java.util.concurrent.Semaphore;

**class** Philosopher3 **extends** Thread

{

**int** id;

Semaphore lfork;

Semaphore rfork;

Philosopher3(**int** id, Semaphore lfork, Semaphore rfork)

{

**this**.id = id;

**this**.lfork = lfork;

**this**.rfork = rfork;

}

**public** **void** run()

{

**try** {

**for**(**int** i = 0; i<100; i++)

{

**if**(**this**.id % 2 == 0)

{

lfork.acquire();

rfork.acquire();

}

**else**

{

rfork.acquire();

lfork.acquire();

}

eating();

rfork.release();

lfork.release();

thinking();

}

}**catch**(InterruptedException e)

{}

}

**void** eating()

{

System.***out***.println("[" + id + "] eating");

}

**void** thinking()

{

System.***out***.println("[" + id + "] thinking");

}

}

**public** **class** philo3 {

**static** **final** **int** ***num*** = 5;

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**int** i;

Semaphore[] fork = **new** Semaphore[***num***];

**for** (i=0; i<***num***; i++)

fork[i] = **new** Semaphore(1);

Philosopher3[] phil = **new** Philosopher3[***num***];

**for** (i=0; i<***num***; i++)

phil[i] = **new** Philosopher3(i, fork[i], fork[(i+1)%***num***]);

**long** sx = System.*currentTimeMillis*();

**for** (i=0; i<***num***; i++)

{

phil[i].start();

}

**for**(i = 0; i<***num***; i++)

phil[i].join();

**long** ex = System.*currentTimeMillis*();

System.***out***.println( (ex - sx) +"ms");

}

}

* 왼쪽과텍스트이(가) 표시된 사진

  자동 생성된 설명 같이 id%2가 0과 같다면, 짝수라는 의미이므로 왼쪽 포크를 먼저 요구하도록 한다.
* 그 외에는 홀수 철학자이므로 우측 포크를 먼저 요구하게 된다.
* 이렇게 되면 식기를 요구하는 순서가 남이 이미 점유하고 있는 것을 먼저 요구하는 방식이 되므로, 스레드가 자원을 점유하지 못한 상태로 wait시켜 환형상태가 없어진다.

1. 해결책들에 대해 설명 및 간단한 비교 평가를 제시하시오.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 메인 함수에는 공통적으로 위와 같은 형식의 현재 시간 측정 타이머 함수가 들어가게 된다. sx로 시간을 잰 후 스레드들을 시작시키고, join으로 스레드들이 종료된다면 그 시간을 ex로 재어 둘의 차를 출력하는 방식으로 성능을 측정할 수 있다.
* 본래 코드에서는 각 스레드당 100번의 루프를 하게 했는데, 실험을 위해 반복횟수를 10000씩으로 늘려 각 해결책들의 시간차를 가시적으로 확인할 수 있게끔 하였다.
* 각 해결책에 대해 10번씩의 수행을 하여 수행 시간의 평균을 낸 결과는 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| 해결 방법 | No hold and wait | Removing circular wait 1 | Removing circular wait 2 |
| 1회 |  |  |  |
| 2회 |  |  |  |
| 3회 |  |  |  |
| 4회 |  |  |  |
| 5회 |  |  |  |
| 6회 |  |  |  |
| 7회 |  |  |  |
| 8회 |  |  |  |
| 9회 |  |  |  |
| 10회 |  |  |  |
| 평균 |  |  |  |

* 결과에 대한 분석은 아래와 같다.

1. 각 해결책의 평균 time cost에 대한 비교
2. No hold wait이 평균 소모 시간이 가장 긴 이유에 대한 추측

* No hold wait의 경우, 모든 프로세스는 왼쪽과 오른쪽 식기를 모두 들 수 있을 경우가 아니면 실행이 진행되지 않는다. 즉 필요한 모든 자원을 할당 받기 위해 너무 오랜 시간을 사용하게 되는 경우가 생긴다.

1. 환형 대기 제거 해결책들 간의 평균 시간 소모량 분석

* 사실상 표본이 너무 적었고, 표본들의 소모 시간의 분포와 평균 소모 시간에 뚜렷한 차이가 보이지 않는다. 따라서 평균값을 통해 성능의 우열을 비교하기에는 무리가 있다고 판단했으며, 이에 대한 분석을 하도록 한다.

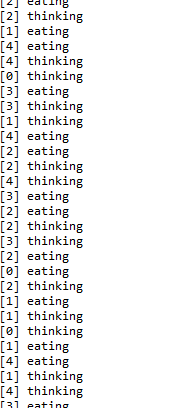
1. 환형 대기 제거 해결책들의 소모 시간 평균 값에 대한 분석

* 두 해결책의 소모시간 분포가 비슷하며, 평균 소모 시간 또한 큰 차이를 보이지 않는다. 그 이유에 대한 추측은 다음과 같다.

1. 실행되는 스레드가 5개에 불과함과 동시에, 환형 대기 조건을 깨기 위해 ordering에 변화를 준 쓰레드 개수는 해결책 1은 1개, 해결책 2는 2개에 불과하다.
2. 프로그램을 실행한 횟수, 즉 표본 개수가 작아 실험 환경에 의한 변수(background programs)가 평균 값에 막대한 영향을 주어 극단값에 의해 값이 오염되었다. 즉, 수집해야할 표본의 개수가 훨씬 더 늘어나야했다.
3. 텍스트이(가) 표시된 사진

   자동 생성된 설명테이블이(가) 표시된 사진

   자동 생성된 설명결론

* 왼쪽부터 no hold and wait, removing circular wait 1, 2의 실행 결과의 중간부이다. 그림에서 알 수 있듯이, 모든 해결책에서 모든 프로세스가 실행되는 것이 출력되었다. 실행결과를 모두 둘러 보았을 때, Deadlock을 발견할 수 없었다.
* Deadlock을 prevent하는 방법으로, 점유대기와 환형대기 조건을 없애는 방법을 알아보았다. 점유 대기의 경우 프로세스에 필요한 모든 자원을 할당하는 방법을 사용했으며, 환형 대기의 경우 할당하는 ordering에 변화를 주어 closed chain을 없애 두 조건을 제거하여 교착상태를 방지할 수 있었다.
* C에 대한 아쉬움이 크다. 실행 횟수가 훨씬 커지고, 극단값이 평균에 영향을 미치는 것을 줄여주기 위해 기존 평균에 가중치를 두었다면 더 신뢰성이 있는 결과가 나왔을 것이라 생각된다.
* 교착 상태를 공부함으로써, 멀티미디어 환경에서 deadlock이 얼마나 critical할 수 있는 가를 추측할 수 있었다.