동시성 프로그래밍[C]

Process : 계산을 실행하는 주체를 가리킨다.

프로세스의 상태

1. 실행전상태: 계산을 싱행하기 전의 상태 실행상태로 전이할 수 있음.
2. 실행상태: 계산을 싱행하는 상태. 대기 상태 또는 계산종료 상태로 전이할 수 있음.
3. 대기상태: 계산을 일시적으로 정지한 상태, 실행상태로 전이 가능.
4. 종료상태: 계산을 종료한 상태를 말함.

프로세스의 대기상태 변화 이유

1. 데이터 도착을 대기한다. -> 종속되는 데이터가 있을 시 프로세스는 해당 데이터가 도착하기를 기다리는 동안 대기상태가 된다.
2. 계산 리소스 확보를 기다린다. -> 뮤텍스를 통한 데이터 사용이 있을 시에 프로세스는 리소스를 사용할 수 있을 때까지 대기한다.
3. 자발적 진입 -> 타이머와 같은 상황을 예로 들 수 있다. 아무것도 수행하지 않아도 될 때 프로세스는 대기상태로 전이한다.

동시성이란?

동시성은 2개 이상의 프로세스가 동시에 계산을 진행하는 상태를 말한다.

즉 프로세스 2개가 같은 시간대에 사이에 작업이 일어나며 동기 계산 또는 상태를 유지할 수 있는 기반을 나타낸다.

벙렬성이란?

병렬성은 같은 시간에서 여러 프로세스가 동시에 계산을 실행하는 상태를 의미한다.

병렬성의 3가지 종류

태스크 병렬성 – 여러 태스크가 동시에 실행되는 것을 의미한다.

데이터 병렬성 – 데이터를 여러 개로 나눠 병렬로 처리하는 방법을 말한다.

인스트럭션 레벨 병렬화 – CPU 명령어 레벨에서 병렬화를 수행하는 방법을 말한다.

[높은 수준의 최적화를 수행할 때는 인스트럭션 레벨 병렬화가 이루어진다]

[계산량이 적은 문제의 경우 병렬성을 도입할 때 오버헤드가 더 커지는 상황이 발생하기 때문에

연산이 더 느리게 될 수 있다.]

* 응답 속도와 처리량

응답속도 = 소비 cpu 클록 수 / cpu 작동 클록 주파수

암달의 법칙 병렬화로 인한 성능향상 확인법

1/1-P+(P/N)

P는 병렬화를 진행한 함수가 수행하는 시간

1 – P는 즉 병렬화를 진행하지 않고 수행하는 시간

N는 병렬화하는 수.

즉 오버헤드 0.5로 치고 병렬화 가능 비율이 0.5의 경우 식은

1/ 0.5+ (1-0.5) + (0.5/4) = 1/1.125 = 0.889

순차 실행보다 1.125배 수행시간이 더 걸리게 되는 상황이며 병렬화가 더 느려진다는 결과

레이스 컨디션[경합상태]

여러 프로세스가 동시에 공유 자원에 접근함에 따라 일어나는 예상치 않은 이상이나 상태를 의미

동시성에서는 이와 같은 레이스 컨디션을 방지하는 것이 중요

예를 들어

A와 B 프로세스 두개가 V라는 integer형 공유자원을 사용할 때 critical section을 설정하지 않을 경우 각 자 2번씩 V값을 증가했을 경우 이 값은 4가 나와야하지만 3이 나오는 등 기대한 값과 다른 값이 나올 수 있다.

Read와 Write 시점에 따라 위와 같은 상황이 발생할 수 있기 때문에 우리는 뮤텍스, 세마포어

Critical\_section 등 공유 자원에 대한 제어를 해줄 필요가 있다.

아토믹 처리

여러 번의 메모리 접근이 필요한 조작이 조합된 처리를 말한다.

처리가 아토믹하다는 “처리 도중 상태를 시스템적으로 관측이 불가하며 처리 실패시에 전 상태로 복원되는 것을 말한다.”

Compare and Swap

Bool C\_S(uint64\_t \*p, uint64\_t val, uint64\_t newval){

If(\*p != val) return false;

P\* = newval;

Return true;

}

동기화 기능에 따른 확인 함수 구현

뮤텍스(Mutual Execution)

배타 실행이라 불리는 동기처리방법으로 크리티컬 섹션에서 실행가능한 프로세스 수를 1개로 제한하는 동기화 방법이다. – 일반적으로 공유변수로 사용할 flag를 준비 후 해당 flag의 true, false

여부에 따라 크리티컬 섹션을 실행한다.

Ex)

Bool lock = false;

Void function(){

Retry:

If(!lock){

Lock = true; // 프로세스가 점유

// 실행할 함수 구문

}else goto retry;

Lock = false; // 락 해제

}

스핀락

위의 코드에서는 루프를 통하여 계속하여 대기를 하였다. 이와 같이 리소스가 비는 것을 기다리며 확인하는 방식을 스핀락이라 부른다.

Void spinlock\_aqcuire(bool \*lock){

While(test\_and\_set(lock));

}

Void spinlock\_release(bool \*lock){

Tas\_release(lock);

}

이와 같은 방식을 사용하면 락 획득이 가능할 때까지 계속해서 공유 변수를 확인하기 때문에

크리티컬 섹션 안에 연산량이 많아지는 경우 불필요한 cpu 리소스를 소비하게 된다.

때문에 락획득을 못하였을 경우 Context switch로 cpu를 명시적으로 전달해 계산자원을 효율하는 방법도 있다.

또한 단일 스핀락의 경우 OS에 의한 할당 제어가 어렵기 때문에 권장하지 않는다.

Pthread 예제

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

Pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER; // mutex 초기화

Void func(void \*arg){

If(pthread\_mutex\_lock(&mut) !=0) exit(-1);

크리티컬 섹션 - 실행구문

If(pthread\_mutex\_unlock(&mut) !=0) exit(-1);

Return null;

}

세마포어

뮤텍스에서는 1개의 프로세스에게만 접근권한을 주어졌지만 세마포어는 n개의 프로세스에 동시에

공유자원에 대한 락을 주는 방식이다.