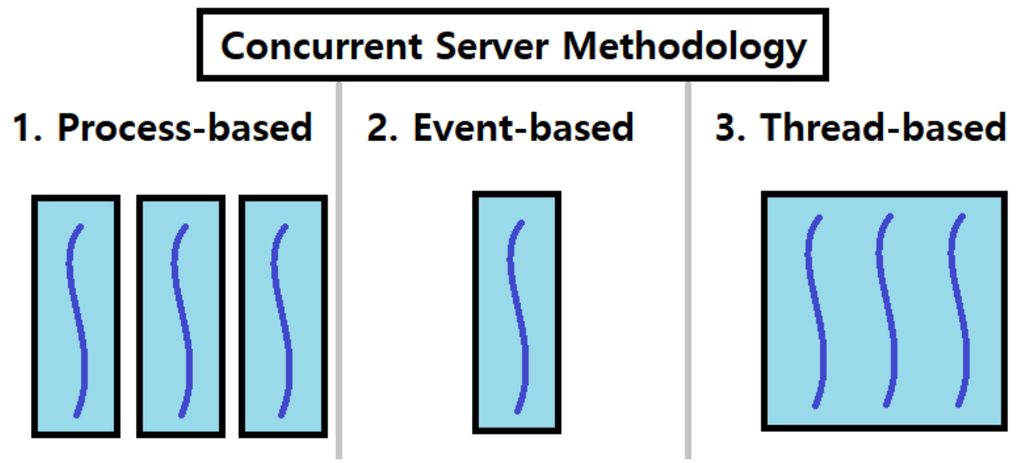
SP - 4.1 Concurrent Programming

hyeok's Log · 2022년 4월 14일

sp

SystemProgramming

목록 보기 12/29



Introduction

Concurrent Programming이란, 앞서 Process와 Signal 개념에서 다룬 Concurrent Flow를 떠올리면 되는데, 여러 프로그램과 Flow가 동시적으로 수행되는 상황을 다루는 프로그래밍을 의미한다.

Concurrent Programming은 기본적으로 어렵다. 왜냐? 인간은 본능적으로 순차적이 프로그래밍에 익숙하기 때문이다. 동시 프로그래밍은 우리가 예측하지 못하는 흐름을 가질 때가 많다. 우리가 일정 부분 이상은 제어할 수 없다. **다양한 Possible Flow를 모두 조심히 다루어야한다.** 그래서 어렵다.

- 우리가 **일반적으로 맞이하는 Cocurrent Programming의 어려움**은 다음과 같다.
    - **Race Problem : 프로그래밍의 결과가 시스템의 임의 스케쥴링 결정에 의해 좌우된다.**
        - ***즉, 스케쥴링 상황에 따라서 프로그램의 결과와 Flow가 달라진다.***
        - fork를 통해 4개의 프로세스를 생성했다고 해보자. Pa~Pd가 바로 그들이다.
        - 이 프로세스들은 **'Key Variable(Live Lock)'을 획득하기 위해 모두가 경쟁**한다.
        - 첫 번째 경쟁에서, Pa가 Winner가 되었다. Pa가 Key Variable을 이용해 동작을 수행하고 나서, 다시 경쟁이 시작된다.
        - Pa~Pd가 똑같이 경쟁을 했는데, 또 다시 Pa가 Winner가 되었다.

- 또 경쟁을 했는데, 역시나 Pa가 이겼다.
- 다시 경쟁을 했는데, 이번엔 Pb가 이겼다. 그 다음 경쟁은 Pc가 이겼고, 마지막 경쟁에선 Pb가 이겼다.
- **Pa~Pd 프로세스가 모두 Key Variable을 갖기 위한 동일한 목표를 가지고 경쟁을 했는데, 매 경쟁마다 누가 이길지를 모른다.**
    - 이 예시에서 **Pd 프로세스는 한 번도 Winner가 되지 못했다.**
        - 이를 **'Starvation' 문제**라고 부른다. 즉, Pd는 굶는것이다.
- 각 프로세스가 **공정하게 경쟁을 시작했음에도, 결과는 그러지 못했다.**
    - 이를 **'Fairness' 문제**라고 부른다. 누군가 독식하는 프로세스가 존재하는 것이다.

> ***Race Problem with Livelock, Starvation, Fairness Problems!***

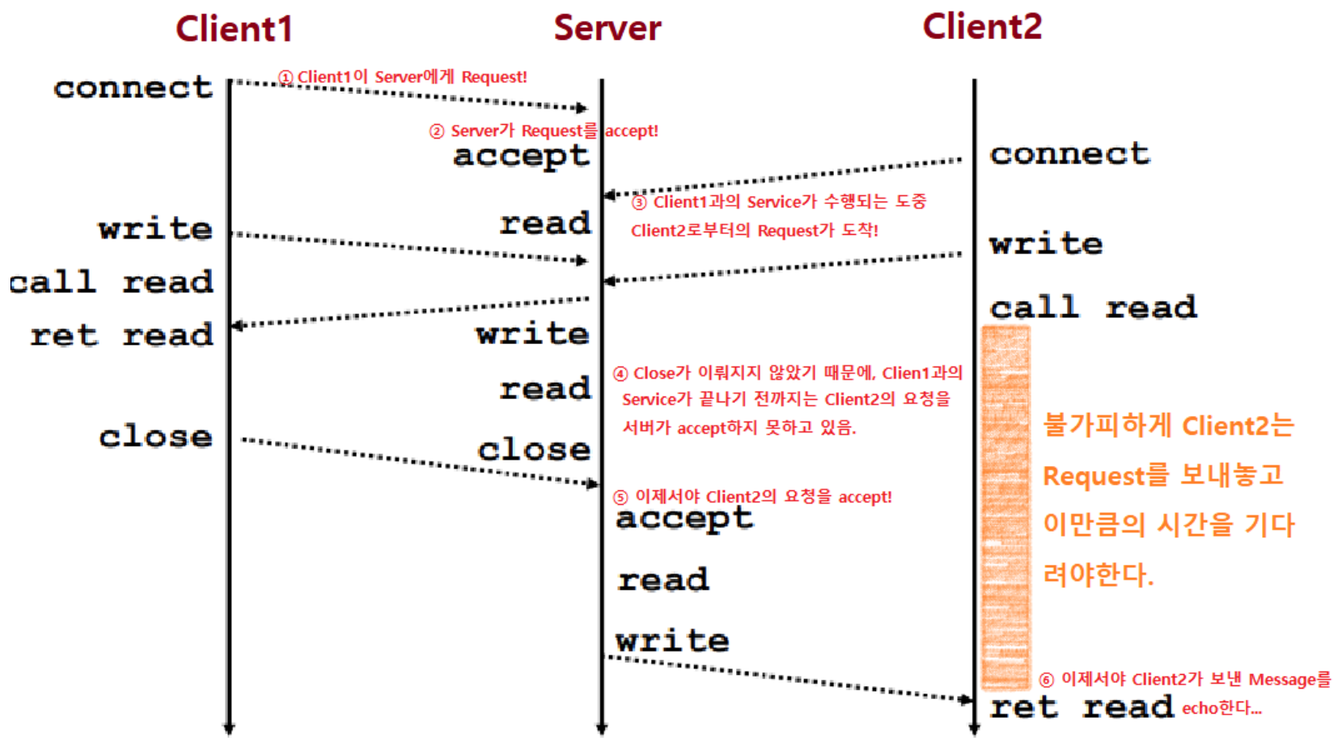
- **Deadlock Problem : 부적절한 Resource Allocation이 앞선 프로세스들의 진행을 막는 현상**
    - **서로 다른 프로세스가 서로의 진행을 기다리는 Hanging Situation**이 바로 Deadlock Problem으로, 앞선 포스팅에서 'Async-Signal-Safety'에 대해 설명할 때, Signal Handler에서 printf를 사용하면 이런 문제가 발생할 수 있다고 언급한 바 있다.
    - Deadlock Problem은 **매우 미묘하고, 가끔 발생하기 때문에 해결하기가 상당히 어렵다.**
- **Shared Resource의 Corruption 문제**가 발생할 수 있다.

> 이처럼, Concurrent Programming은 **미묘한 Race, Livelock, Starvation, Fairness, Deadlock 문제가 발생할 수 있다.**

> **그래서 어렵다.** 각 Problem들의 상세 개념은 나중에 OS(운영체제) 포스팅에서 자세히 설명하겠다.

Problem of 'Not Concurrent Server'

앞선 네트워크 프로그래밍 포스팅에서 우리는 Echo Server를 구현했다. 이 서버는 **'Not Concurrent Server(Iterative Server)'**라고 했다. 이 예시를 통해 우리는 Concurrent Programming의 필요성을 느낄 수 있다. 아래 그림을 보자.



- 서버가 Launch하고, Client1과 Client2가 서버에게 Connection 요청을 한 상황이다. **Client1이 스케쥴링 상 먼저 connect를 요청하게 되어 서버가 이를 accept**하였다.
    - **서버와 Client1이 Connection을 맺고, clientfd와 connfd 사이에 Channel이 형성**되었다.

- 한편, **Client2도 거의 같은 시간에 Connection을 Request했지만, 대기하게 된다. 예시 Echo Server는 Iterative하게 동작하고 있기 때문**이다.
    - ***앞선 Connection이 끊기기 전까진 새로운 accept를 하지 못한다.*** ★★
- Client1이 "boy"라고 write하면, **서버 프로세스 쪽 OS 내의 Network Stack's TCP Manager(Stack)**에 "boy"가 쌓인다.
    - 서버는 바로 이를 출력한다.
- **Client2는 Server와 Connection이 맺어지진 않았지만, Server에게 Message를 write할 수는 있다. (connect는 Request만 보내고 바로 수행이 끝난다. 그말은 즉슨, 반대편에서 Request를 받든 말든, 일단 동작을 할 수 있는 것이다. ★★★)**
    - Client2는 **"girl"**이라는 메세지를 보낸다.
- 서버가 Client1에게만 Service를 제공하고 있으므로, **"girl"은 TCP/IP Stack에 쌓이기만 할 뿐, 나가질 못한다. 대기하는 것**이다.
- **서버와 Client1의 Connection이 끝나면, 그제서야 Client2의 Request를 accept하고, 그제서야 "girl"이라는 메세지를 뽑아내서 읽고 write할 수 있다.**

> 문제가 무엇인가?
>
> > 그렇다. **Connection을 빌드하지 못한 Client는 무작정 대기**해야하는 것이다.
> >
> > > 서버 본연의 역할을 제대로 수행하지 못한다. **(Iterative Server의 문제점)**

- **connect function의 이해**

    - **Client가 connect 함수를 통해 Connection Request를 보냈을때, 서버에서 이를 accept하지 않아도, Client는 데이터 통신을 시도할 수 있다.**
        - ***서버 쪽의 OS 커널 내 Network Stack에 있는 TCP Manager에 Request가 큐잉된다. accept는 되지 않고 말이다.***
            - 이런 특징을 **'TCP listen backlog'**라고 부른다. ★

- **Request는 보냈는데, accept가 이뤄지지 않은 상황에서, Client가 Data를 보내면, 서버 쪽의 TCP Manager에 데이터가 버퍼링**된다.

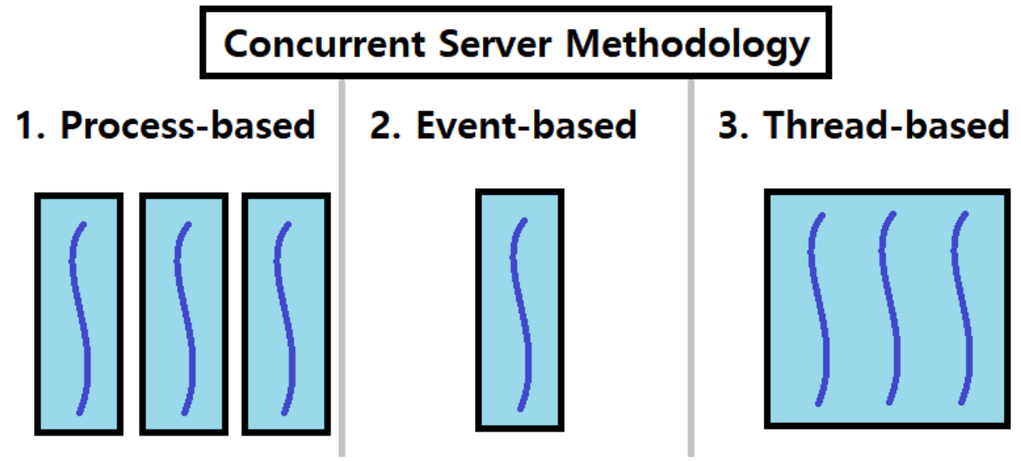
- 즉, 위의 Echo Server 예시 기준으로, Client2 쪽의 connect와 rio_writen 함수는 리턴을 마친 것이다.
    - 그러나, 양쪽의 rio_readlineb는 blocking되어 있는 것이다.

- **서버는, Connection이 빌드된 Client를 제외하고는, 다른 Client의 데이터를 Consume하지 못하고 있는 것**이다.

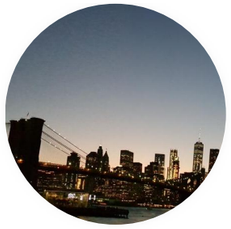
> **Iterative Server 대신 Concurrent Server를 사용해야한다.**
>
> > ***Concurrent Server는 여러개의 Concurrent Flow를 이용해 여러개의 Client에게 동시에 서비스를 제공한다.*** ★

Concurrent Server 방법론

Concurrent Server를 구축하는 방법은 크게 3가지가 있다. 우선, 자세한 설명에 앞서 아래의 그림을 보자. 각 방법론을 아주 적절하게 표현한 그림이다.



- **Process-based : 서버 프로세스에서 fork를 이용해 여러 복제 프로세스를 띄우는 방식**이다.
    - **Kernel은 자동적으로 복수의 Logical Flow를 운영**한다.
    - **하나의 프로세스가 하나의 Flow**를 나타낸다. 프로세스를 띄우는 방식이므로, **각각의 Private Address Space가 존재**한다.
- **Event-based : 하나의 프로세스에서 일련의 이벤트 처리 루틴을 이용해 동시성을 제공**하는 방식이다.
    - **프로그래머가 직접 복수의 Logical Flow를 운영**한다.
    - 모든 Flow가 하나의 프로세스 위에 존재하므로, **동일한 Address Space를 공유**한다.
    - **I/O Multiplexing이라는 기술을 사용**한다. ★
- **Thread-based : 하나의 프로세스에서 여러 Thread를 만들어, 여러 Execution Flow를 운영**하는 방식이다.
    - **Kernel은 자동적으로 복수의 Logical Flow를 운영한다. (Just like Process-based)**
    - **모든 Flow가 공통된 Address Space를 공유**한다.
        - 이는 추후 Thread 개념을 소개하면 무슨말인지 이해할 수 있다.
    - **Process-based와 Event-based의 속성이 합쳐진 Hybrid 방식**이라 할 수 있다.



hyeok's Log

팔로우

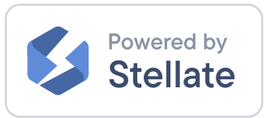
이전 포스트
SP - 3.3 Socket Interface - Ech...

다음 포스트
SP - 4.2 Process / Event-base...

0개의 댓글

댓글을 작성하세요

댓글 작성


Powered by
Stellate