SP - 4.2 Process / Event-based Server

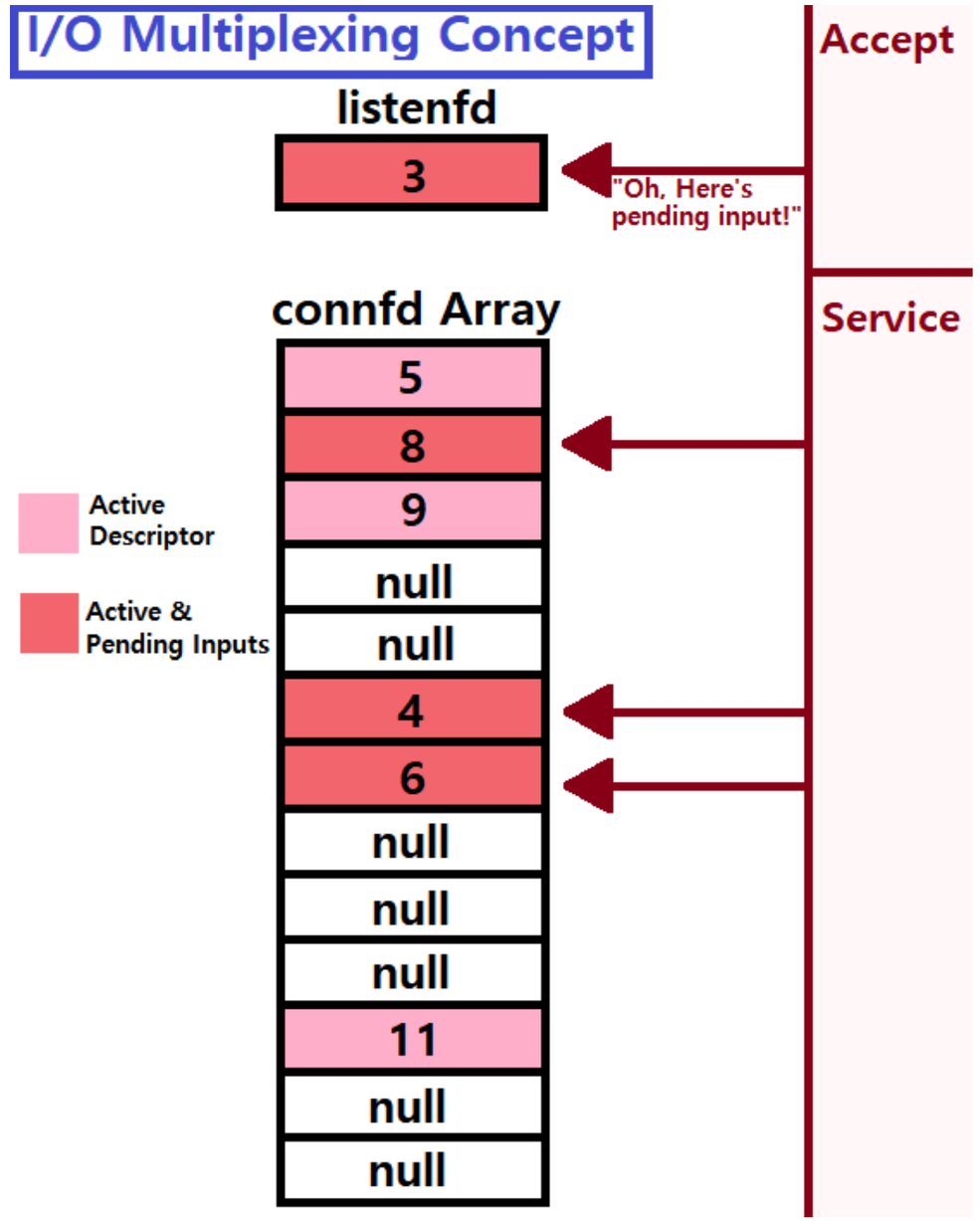
hyeok's Log · 2022년 4월 15일

팔로우

sp

1

SystemProgramming

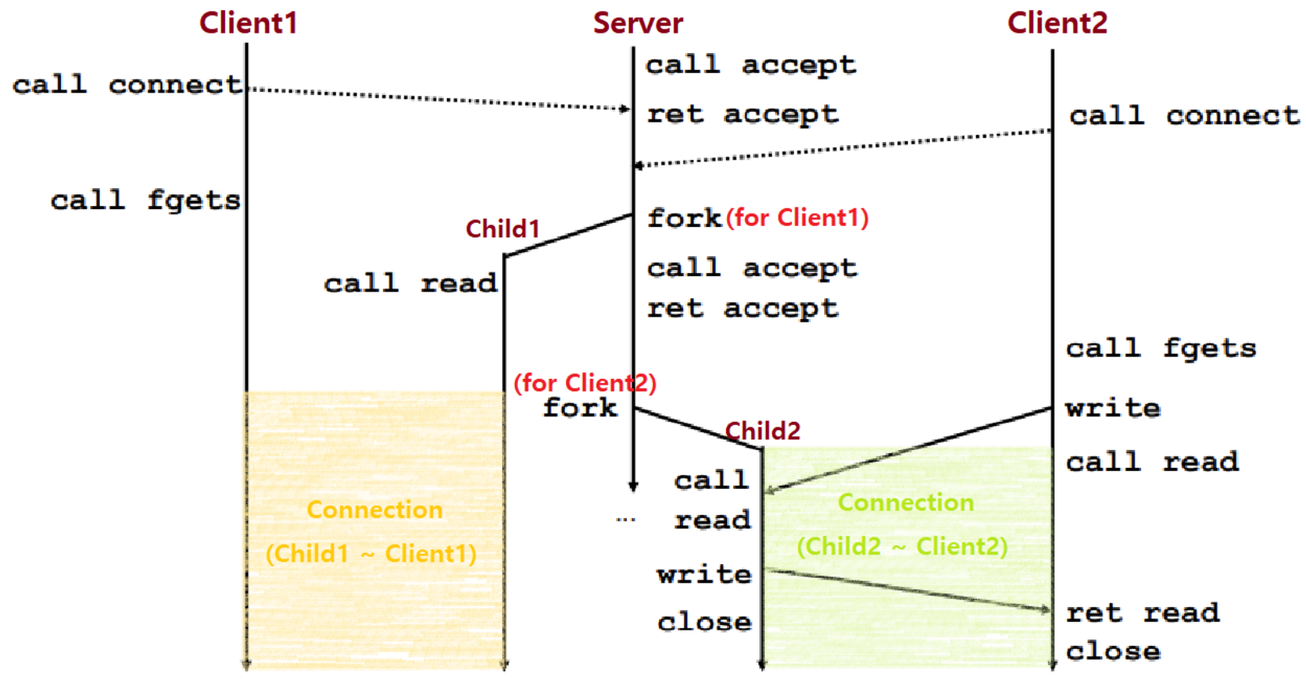

I/O Multiplexing Concept
listenfd
3
"Oh, Here's pending input!"
Accept
Service
connfd Array
5
8
9
null
null
4
6
null
null
null
11
null
null
Active Descriptor
Active & Pending Inputs

Process-based Concurrent Server

Conceptual Details

> **Process-based : 각 Client와의 Connection을 위해 '분리된 프로세스'를 생성**한다.

- Process-based Concurrent Server는 **fork가 핵심**이다.
- **Client1이 Connection Request를 보내면, Server가 accept**를 한다.
    - ***accept return 후, 바로 fork를 하여, Server의 Child Process와 Client1이 Channel을 형성하게 만든다.*** ★
        - 즉, **Child와 Client가 Connection을 맺는다.**
- 즉, fork로 생성된 **Server의 Child Process에게 connfd File Descriptor를 넘겨주는 방식**이다.
- **Parent Process(Server)가 accept를 호출하고, 반환 디스크립터를 받으면, 바로 fork를 한다.**
    - fork는, 알다시피, **Parent를 그대로 복사해서 Child를 만든다.**
- fork 이후, Parent는 그대로 다시 accept를 수행할 수 있게 프로그래밍한다. listen은 이미 fork 이전에 해놓은 상태이니까 당연히 가능하다.



- **Client N개가 서버에게 Request를 보내면, N개의 Child Process가 생성되는 것이다. ★**
- fork시, 새로 생성된 프로세스는 **Parent Process의 데이터를 복사하긴 하지만, Address Space 자체는 별도로 독립적으로 존재**하므로, 마찬가지로, 위의 상황에서도, **Child Process Server들이 모두 별도의 Address Space를 점유**한다.

> **accept 이후 close가 이루어져야 다시 accept를 할 수 있다**고 했다. 그래서 Iterative Server가 문제를 보이는 것이었다.
>
> > **따라서, Process-based Server에서는, accept-fork 이후, (Parent에서) 바로 해당 connfd를 close한다.**
> >
> > > ***close를 하더라도, Child에는 영향이 없기 때문이다.***

- 이때, 만약, accept 이후, close 이전, 또는 close는 했지만, 아직 Loop 한 차례 순회가 돌기 이전에, 그 미묘한 시간에 다른 Client로부터 Request가 오면 어떻게 하는가?
    - 걱정할 필요 없다. **OS Kernel의 Network Stack 위치에 있는 TCP Manager**에 Request가 큐잉된다고 하지 않았나. 따라서 알아서 처리된다.

Implementation Details

아래는 Process-based Concurrent Server의 기본 코드이다.

```
/* fork로 생성된 Child의 서비스가 어느 순간 종료되면, Reaping! */
void sigchld_handler(int sig) {          // errno 임시 보관, Signal Masking 등을
    while (waitpid(-1, 0, WNOHANG) > 0) // 하지 않은 Naive한 SIGCHLD Handler이다.
        ;
    return;
}

int main(int argc, char **argv) {
    socklen_t clientlen;
    int listenfd, connfd;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);       // Child Reaping을 위한 핸들러 설치
    listenfd = Open_listenfd(argv[1]);      // listen까지의 작업 수행
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);        // 중요한 부분
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);  // Accept!

        /* Accept 이후 Child Server Process 생성 */
        if (Fork() == 0) {
            Close(listenfd);    // Child가 Request를 빼앗지 않도록 listen 종료 ★★★
            echo(connfd);       // Child의 Service 제공
            Close(connfd);      // Child 서비스 종료되면 해당 디스크립터 close
            exit(0);            // Child Termination -> SIGCHLD Handler will catch
        }
        /* Parent Server Process에서는 fork 이후 바로 connfd를 close! */
        Close(connfd);          // 매우 중요한 부분. 계속 accept해야하므로! ★★★
    }
}
```

- **fork 시 Parent(Server)가 가지고 있는 속성들은 모두 Child Server Process로 복사**된다.

    - 만약, 예를 들어, Parent Process에서 listenfd가 File Descriptor Table 상 3번 인덱스였다고 해보자.
        - 그렇다면, accept 함수의 리턴값인 connfd는 4번을 의미할 것이다.
            - 이 **File Descriptor가 그대로 Child에게도 상속**된다.

- Child에선 listenfd를 바로 close하고, connfd를 이용해 Service를 제공한다. 그 다음, connfd를 close한다.

    - 이때, **Child에서 바로 listenfd를 close하는 이유**를 알아야한다.

> ***다른 Client가 Server에게 Connection Request를 보내면, Server(Parent)가 받아야하는데, 이를 Child가 받을 수 있기 때문이다. 그래서, Child가 생성되자마자 바로 해야할 일은 물려받은 listenfd를 닫는 것이다.*** ★

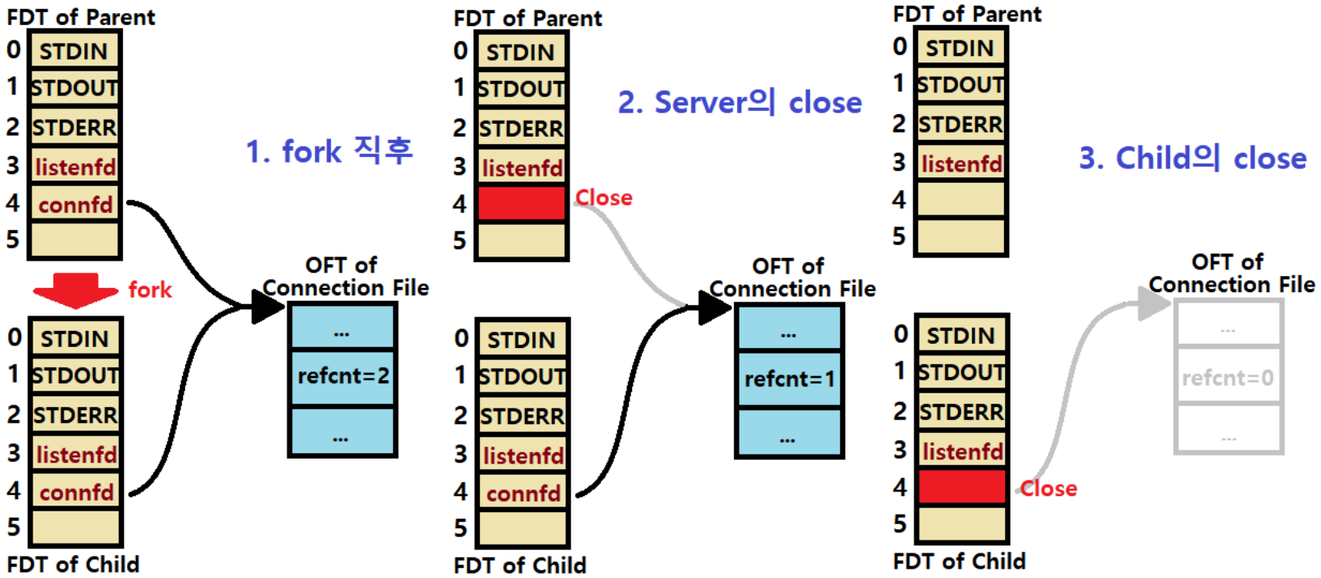
- 이때, **Parent는 fork 이후 바로 connfd를 close해야한다. 왜냐? 앞서 말했듯이, accept를 또 해야하기 때문**이다. ★

- fork 직후 시점을 다시 생각해보자. **connfd가 가리키는 File에 대한 Open File Table의 refcnt값은, 해당 File을 가리키는 프로세스가 2개가 되었으므로 1에서 2로 바뀔 것**이다. ★

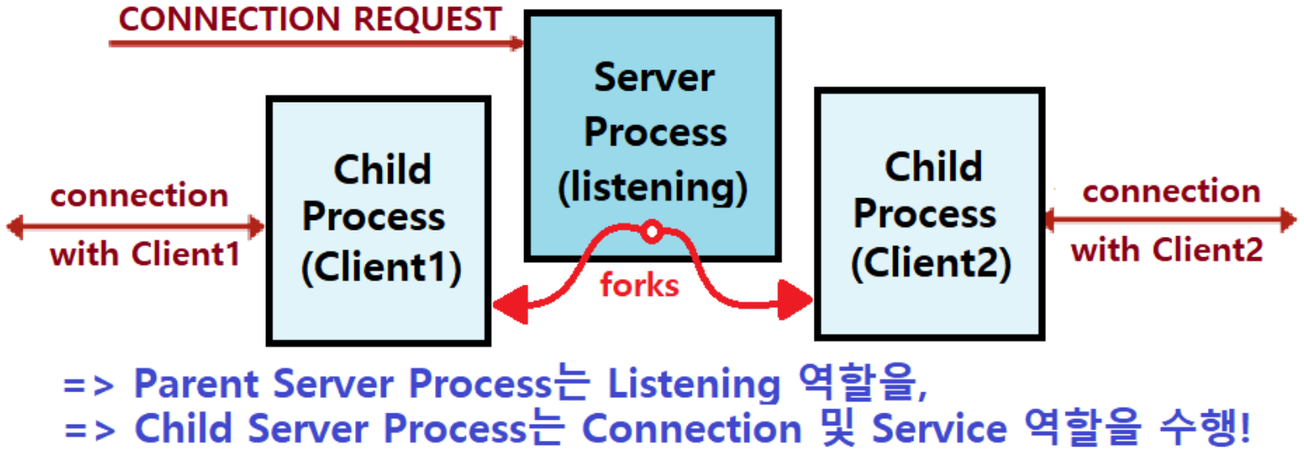
    - **fork 이후, Parent가 close를 하므로, connfd file의 refcnt는 이어서 바로 다시 1로 바뀔 것**이다.
        - 이후, **Child Process와 Client의 Connection이 종료되면, refcnt가 0이 되고, 해당 Connection File이 소멸**된다. ★

~> 이와 같은 처리를 통해 각 Child Process가 가리키는 **Connection File이 매번 지워져야 Memory Leakage를 방지**할 수 있다.

아래 그림을 보며 이해하자.



- Process-based의 accept 과정을 다시 한 번 상세히 설명하겠다.
    - **서버는 accept에서 블로킹되어있다. Request가 올때까지 말이다. Listening File Descriptor listenfd에서 말이다.**
- **Client에서 connect 함수를 호출해 Connection Request를 보내면, 서버는 accept를 통해 받아들이고, connfd를 만든다.**
    - 이후, **fork를 하여, 그 connfd와 clientfd 사이에 Connection을 맺고 Service를 수행**하는 것이다.
- 와중에 **Parent는 fork 이후 바로 connfd를 close하고, 새로운 Request를 기다리게 되는것!**



- **각각의 Client는 서로 다른 독립적인 Child Process에 의해 핸들링**되고 있다.
- **각 Child끼리 어떤 공간이나 상태도 공유하지 않는다. ★**

> Parent와 Child는 동일한, 복사된 listenfd와 connfd를 갖는데,
>
> **Parent는 fork 이후 바로 connfd를 close해야하고,**
> **Child는 fork 이후 바로 listenfd를 close해야한다.**

- Process-based Server 구현 시 주의점
  - **Listening Server(Parent)는 반드시 Child Termination 시의 Zombie Process를 Reaping해야한다.**
    - 치명적인 메모리 누수를 방지하기 위해!
      - *by SIGCHLD Signal Handler (Look above!)*
  - **Parent는 반드시 connfd 복사본을 close해야한다.**
    - **OS Kernel은 Socket/Open File에 대한 refcnt를 기억**하고 있다. (Open File Table,

Entry)

- **fork 이후, refcnt는 2가 되기 때문에, 이 refcnt를 0으로 만들어주어야 한다. 그래야만 Connection이 제대로 사라지는 것이다. ★**

Pros and Cons

- **장점**
  - **여러 Connection을 동시적으로(Concurrently) 처리**할 수 있다.
  - **Open File Table을 제외하고는 다른 것은 공유하지 않기 때문에 깔끔한 Sharing Model**을 가질 수 있다. (Clean Sharing Model)
    - ***File Descriptor 비공유 (바로 close하니까)***
    - ***Open File Table 공유***
    - ***각 종 Variables 비공유***
  - 이론이 **단순하고 직관적**이다. (쉽다)

- **단점**
  - 기본적으로 **fork를 해야하기 때문에 Process Control을 위한 Overhead가 많다.** (시간적인 오버헤드가 크다)
    - 각 프로세스마다 별도의 공간이 필요하므로 공간적인 Overhead도 크다.
  - **Process끼리 Data Sharing이 없으므로, 오로지 IPC(Inter Process Communication)를 통해서만 각 Process가 통신**할 수 있다.
    - FIFO 형태의 파이프라인 같은 것 말이다! (Shell Program을 생각하자)

Event-based Concurrent Server

I/O Multiplexing Concept

단일 프로세스를 가지고 두 종류의 이벤트를 처리하는 서버를 만든다고 해보자. 서비스는 그대로 'Echo'로 하자.

- 이 프로세스(서버)가 **다뤄야하는 이벤트**는 다음과 같다.
  - Network **Client가 Connection Request를 서버에** 날린다.
  - **User가 키보드로 메세지, 명령 라인을 타이핑**한다.

서버는 이벤트1을 accept해야한다. 동시에 서버는 이벤트2도 핸들링해야한다. 사용자가 키보드에 타이핑하는 것에 응답(반사)해야한다.
~> *이를 단일 프로세스로, fork없이 어떻게 해야할까?*

> 서버의 고민 : 어떤 이벤트를 먼저 기다려야할까?

- 단일 프로세스이므로 병렬처리가 안되므로, 순차적인 핸들링을 시도해볼 수 있다. 예를 들어, Event1부터 처리하고, Event2를 처리하고 하는 식으로 말이다.
  - 하지만, 이렇게 구성할 경우, Event1이 오지 않으면 Event2를 처리할 수 없다. 계속 기다리게 된다.

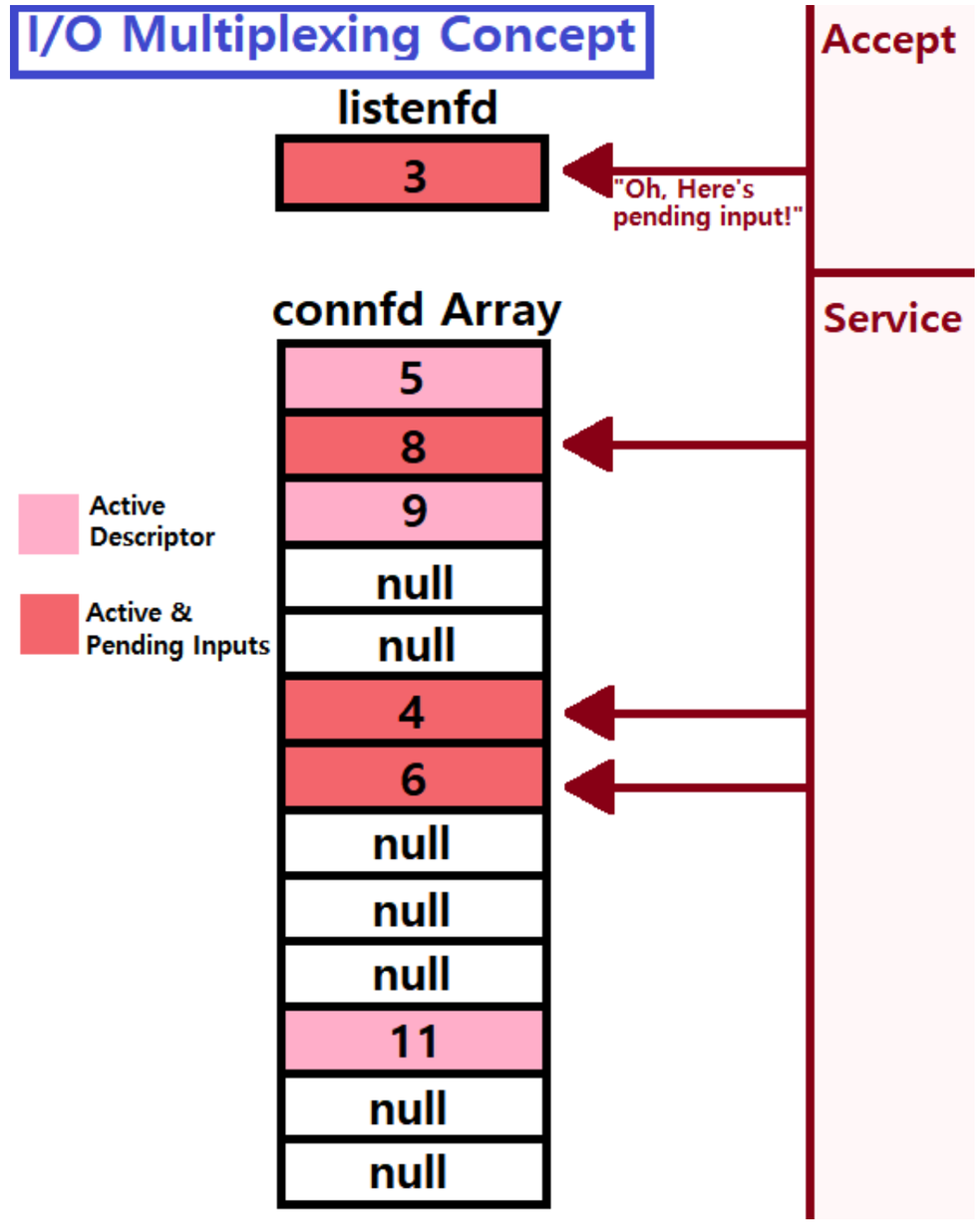
> 그래서 등장한 개념이 **'I/O Multiplexing'**이다.

- **I/O Multiplexing**
  - **Server는 'Active Connection'에 대한 집합을 만든다.** 어떻게?
    - ***By 'Array of connfds' : connfd를 배열로 만든다.*** ★

> **Server가 connfd Array를 돌면서 '어떤 Descriptor(listenfd나 connfd)에 Pending Input이 존재하는지'를 지속적으로 체크한다.**
>
> > *이때, Pending Input은 곧 '이벤트가 도달했음'을 의미한다.*

- **select 또는 epoll이라는 함수를 통해 'Pending Input 여부'를 확인**할 수 있다.
- 만약, **listenfd에 Pending Input이 있으면, Connection accept를 수행**한다.
  - *array에 새로운 connfd를 추가한다.*
- 만약, **array의 element, 즉, connfd_i에 Pending Input이 있으면, 그 connfd 데이터 통신, 서비스를 처리**한다.

> 즉, **'I/O Multiplexing'이란, listenfd와 connfd array를 지속적으로 체크하면서 listenfd쪽에 Pending Input이 있으면 accept를 하고, connfd_i쪽에 Pending Input이 있으면 서비스 제공을 하는, 그런 매커니즘**이다. ★★★



Implementation Details

본격적으로 **Event-based Concurrent Server**에 대해 설명하겠다. Code-Level로 Detail을 소개할 것이기 때문에, 코드의 주석 설명을 천천히 음미하도록 하자.

> **select 또는 epoll 함수를 통해 Kernel에게 '프로세스를 잠시 멈추고, I/O Events가 발생하면 다시**

> 프로세스로 돌아가게 해줘'라는 명령을 내릴 수 있다. ★

```
int select(int n, fd_set *fdset, NULL, NULL, NULL);
/* Active(Ready) Descriptor의 개수를 반환한다. 없으면 -1(에러)이다. */

// FD_ZERO : fdset에 있는 모든 비트를 0으로 처리한다.
// FD_CLR : fdset의 특정 k번 비트(fd)를 Clear한다.
// FD_SET : fdset의 특정 k번 비트(fd)를 Set(Turn On)한다.
// FD_ISSET : fdset의 특정 k번 비트(fd)가 Set인지 확인한다.
//   ㄴ> FD_ISSET은 Pending Input 확인 시 유용하게 쓰인다. ★
```

- ***fdset : Bit Vector이다. 모든 비트는 처음에 0으로 되어있다.**
  - *만약, stdin(0번)에 1을 Set하고, listenfd(3번)에 1을 Set하면, 0번과 3번에 Pending Input이 생기는지 확인할 수 있다. ★*

```
void read_cmdline(void) {
    char buf[MAXLINE];
    Fgets(buf, MAXLINE, stdin);
    printf("%s", buf);
}

/* Server with I/O Multiplexing (Event-based Concurrent Server) */
int main(int argc, char **argv) {
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    fd_set read_set, ready_set;          // fd_set Type의 두 비트 벡터를 만든다.

    if (argc != 2)
        unix_error("Usage Error!\n");

    listenfd = Open_listenfd(argv[1]);  // listen까지의 작업 수행. 3번이 넘어왔다고 하자.

    /* 서버 입장에서, stdin과 listenfd의 Pending Input을 확인할 준비를 한다. */
    FD_ZERO(&read_set);                  // read_set은 모두 FALSE로 초기화하자.
    FD_SET(STDIN_FILENO, &read_set);     // read_set에 stdin을 활성화한다.
    FD_SET(listenfd, &read_set);         // read_set에 listenfd도 활성화한다.

    while(1) {
        ready_set = read_set;            // ready_set은 read_set을 카피한다.
        /* ready_set에서 TRUE로 Pending Input이 있는 fd를 Check한다. */
```

```
            Select(listenfd + 1, &ready_set, NULL, NULL, NULL);

            if (FD_ISSET(STDIN_FILENO, &ready_set))
                read_cmdline();
            if (FD_ISSET(listenfd, &ready_set)) {
                clientlen = sizeof(struct sockaddr_storage);
                connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
                echo(connfd);
                Close(connfd);
            }
        }
    }
```

- stdin에 대한 처리가 왜 있는 것일까?
    - 에코 서버 자체에서의 키보드 타이핑을 처리하려는 것이다. 클라이언트에서의 입력 뿐만 아니라, 서버 자체에서의 입력도 처리하려는 것임. 그렇게 중요한 정보는 아니다. 단지, 이벤트 두 개를 동시에 처리하는 것을 익히고자 함이다.

> **ready_set과 read_set을 따로 두는 이유**
>
> **read_set만을 그대로 Select에서 사용할 경우, Set의 값이 바뀌었을 때, 이전의 정보를 기억할 수가 없다. 따라서 ready_set이라는, read_set의 카피본을 만들어서 사용하는 것이다.**
> ★

그런데, 위의 상기한 코드는 문제가 있다.

> 만약에, while문 순회에서, listenfd에만 Pending Input이 있어서 두 번째 if문으로 들어갔다고 해보자.
>
> > echo Service를 하는데, **echo 함수는 이전 포스팅에서 코드를 봤다시피, EOF가 입력될 때까지 수행**된다.

> 이말은 즉슨, **echo 함수 처리가 오래걸리거나, 또는 EOF가 도달하지 않으면, 즉, 끝나지 않으면, 서버는 계속 기다리게 되는 것이다. 기본적으로 이 서버는 Iterative, 단일 프로세스 서버이기 때문**이다. ★★★

- 즉, **Client가 서버에 서비스를 요청해놓고선, Connection 이후 EOF를 보내지 않으면 서버가 마비되는 것**이다. (이런 류의 공격이 가능)
- 이를 해결하기 위해선 어떻게 해야할까?
    - 몇 가지 다양한 방법이 있는데, 한 가지 예를 들자면, **echo 함수가 한 줄 단위로만 처리하게 하는 것이다. EOF 여부와 상관없이 오로지 한 줄만 읽고 뱉게 하는 것**이다.
        - 이를 **'Multiplex at a finer granualarity'**라고 한다. ★

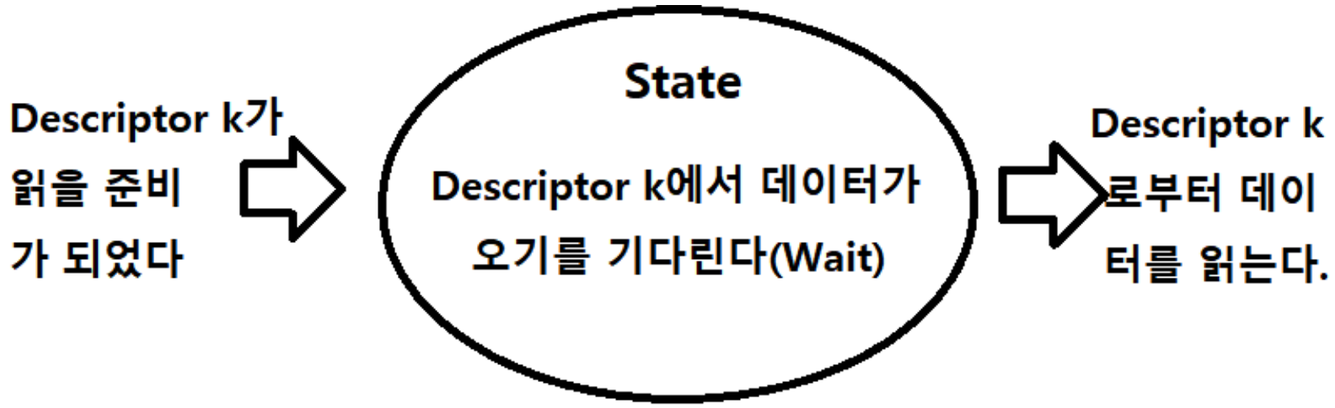
> 위와 같은 문제가 없는, **안정적인 Event-based Concurrent Server를 'Finer Granualarity'가 있는 서버라고 한다.** 이는 프로그래머의 역할이고, 후술한다.

Modeling Logical Flows as State Machines

지금까지 계속 설명한 'I/O Multiplexed Event Processing' 기법은 단일 프로세스가 **어떠한 Flow(짧게 짧게 처리하며 반복하는)를 이용해 Concurrency를 얻는 기법**이다.

> ***Event-based Server = I/O Multiplexing & Event-Driven Programs***

- 이때, **Logical Flow를 하나의 State Machine으로 바라보는 스타일이 존재**한다.
    - 'State Machine for a Logical Flow in a Concurrent Event-Driven Server'




```c
/* Logical Flow를 State Machine으로 바라보기 위한 구조체 */
typedef struct {
    int maxfd;
    fd_set read_set;            // Active Descriptor로 된 비트 벡터
    fd_set ready_set;           // read_set의 부분집합
    int nready;                 // Pending Input이 있는 fd의 개수
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];// RIO Package 사용 시의 버퍼
} pool;                         // pool이라는 이름의 구조체로 정의

void init_pool(int listenfd, pool *p) {
    p->maxi = -1;
    for (int i = 0; i < FD_SETSIZE; i++)
        p->clientfd[i] = -1;    // clientfd를 모두 -1로 초기화

    p->maxfd = listenfd;        // maxfd값은 listenfd값으로 설정
    FD_ZERO(&p->read_set);      // Bit Vector도 모두 0으로 초기화
    FD_SET(listenfd, &p->read_set); // listenfd는 Active로 설정
}

void add_client(int, pool*);
void check_client(pool*);

int main(int argc, char **argv) {
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;

    if (argc != 2)
        unix_error("Usage Error!\n");
```

```
    listenfd = Open_listenfd(argv[1]);      // listen까지의 작업 수행
    init_pool(listenfd, &pool);             // pool 구조체를 초기화

    while(1) {
        pool.ready_set = pool.read_set;     // read_set을 Copy
        pool.nready = Select(pool.maxfd + 1, &pool.ready_set,
            NULL, NULL, NULL);              // Pending Input 중인 fd 개수 반환

        if (FD_ISSET(listenfd, &pool.ready_set)) {  // listenfd에 Event 있으면
            clientlen = sizeof(struct sockaddr_storage);    // (중요 작업)
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);// Accept!
            add_client(connfd, &pool);      // 생성된 connfd를 pool에 업데이트
        }                                   // else는 굳이 하지 않음. 매번 확인하려고

        check_client(&pool);                // 모든 connfd에서 이벤트 발생 여부 확인
    }
}

void add_client(int connfd, pool *p) {
    int i;
    p->nready--;                            // Pending Input 중인 fd 하나 처리했으므로

    for (i = 0; i < FD_SETSIZE; i++) {      // fdset을 쭉 순회하면서
        if (p->clientfd[i] < 0) {           // 비어있는 Array 위치에
            p->clientfd[i] = connfd;        // connfd를 삽입
            Rio_readinitb(&p->clientrio[i], connfd);// 도중에 입력된 데이터 받기

            FD_SET(connfd, &p->read_set);   // connfd가 active한 상태로!

            if (connfd > p->maxfd)          // File Descriptor Table 상에서
                p->maxfd = connfd;          // 가장 인덱스가 큰 connfd인 경우, Update!
            if (i > p->maxi)                // connfd Array에서 가장 큰 인덱스
                p->maxi = i;                // 인 경우에도 Update!

            break;
        }
    }

    if (i == FD_SETSIZE)
        app_error("Error in add_client!\n");
}

void check_client(pool *p) {
    int n, connfd;
    char buf[MAXLINE];
    rio_t rio;
```

Accept vs Connect

```
        // nready가 남아있는 경우, connfd Array를 싹악 훑는다.
        for (int i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
            connfd = p->clientfd[i];
            rio = p->clientrio[i];

            // 현재 조회중인 connfd에 pending Input이 있다면
            if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
                if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {     // 읽어들이고
                    printf("Server received %d bytes on fd %d\n",n, connfd); // 출력
                    Rio_writen(connfd, buf, n);                         // 및 Write
                }
                else {                                                  // connfd에서 EOF가 오면 끊어
                    Close(connfd);                                      // 디스크립터를 닫고
                    FD_CLR(connfd, &p->read_set);                       // fd_set에서 Inactive 만들고
                    p->clientfd[i] = -1;                                // connfd Array에서도 제거!
                }
            }
        }
    }
```

~> 앞서 다룬 예제 프로그램보다 조금 더 형식화되어있고, 운영이 용이한 프로그램이 완성되었음을 알 수 있다.

~> 이런 방식을 'State Machine'화 한 Event-based Concurrent Server라고 한다.

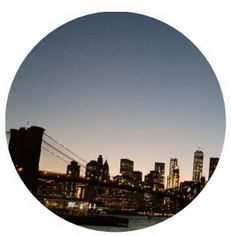
Pros and Cons

- **장점**
  - **하나의 Logical Control Flow와 Address Space로 Concurrent Server를 구동**할 수 있다.
    - 즉, **단일 프로세스로 처리하기 때문에 시공간적인 Overhead가 나머지 두 방법론에 비해서 현저히 적다.**
      - 따라서, Performance가 좋기 때문에 웹 서버나 검색 엔진에서 이러한 구조를 많이 가진다. **ex)** Node.js, nginx, Tornado 등
  - **디버깅이 쉽다. 기본적으로 Top-Down 형태의 Single Step으로 구조를 갖기 때문**
- **단점**
  - 기본적으로 나머지 두 방법론에 비해서 **코드가 복잡**하다.

- **프로그래머가 직접 Fine-Grained Concurrency를 실현해야하기 때문에 프로그래머의 실력이 중요**하다.
- **단일 프로세스이기 때문에 단일 CPU에서 돌아간다.**
  - 그말은 즉슨, 여러 CPU를 사용할 수 없어서 **Multicore의 장점을 사용할 수 없다.**
    - 나머지 CPU가 놀게 되는 것이다. (Single Thread of Control)

Question) fdset의 Implementation Detail이 궁금합니다.
Answer) read_set이 초기에 '00000000'이었다고 해보자.
~> 만약 listenfd(3번이라 가정)와 하나의 connfd(6번이라 가정)가 Active로 설정된다고 하면,
=> '00010010'이 되는 것이다. ★
---> TRUE가 Setting된 fd들에 대해서 Select함수가 Pending Input 여부를 확인하는 것이다.

금일 포스팅은 여기까지이다. 다음 포스팅에선, 남은 하나의 방법론인 'Thread-based Concurrent Server'에 대해 설명한다.



hyeok's Log

팔로우

이전 포스트
SP - 4.1 Concurrent Programm...

다음 포스트
SP - 4.3 Thread-based Concur...

0개의 댓글

댓글을 작성하세요

댓글 작성

Powered by Stellate