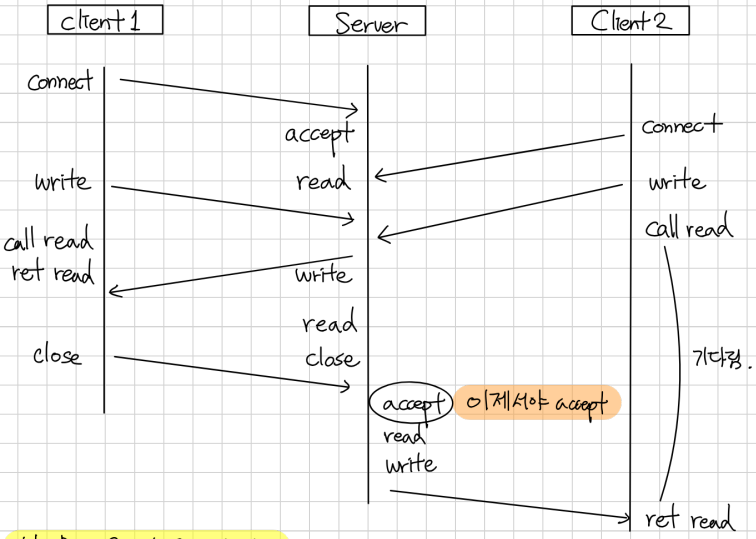
05/13

Concurrent Programming

Concurrent Programming is hard → 왜 어렵나?

- **Races**
  → 결과가 임의의 scheduling 결정에 의해 좌우됨.
- **Deadlock**
  → 부적절한 resource allocation이 프로세스의 진행 막음.
  → 서로 다른 프로세스가 서로의 진행을 기다림. ex) signal handler의 printf.
- **Livelock / Starvation / Fairness**
  → 바쁘게 움직이다가 실패? 상태는 변화하는데 진전이 X

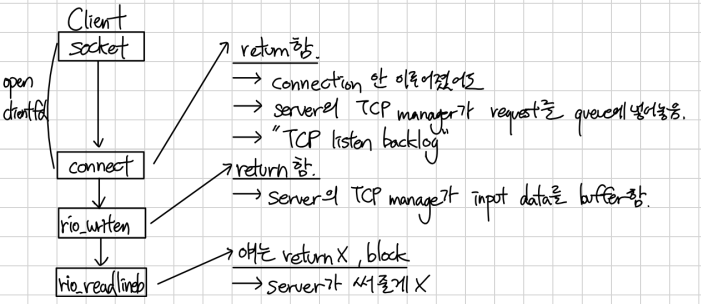
Problem of Iterative Servers (Not Concurrent Servers)



client 2의 경우
connection이 뱉어지진 않았지만,
message를 write은 가능.
↓
server의 TCP manager에
request가 쌓임.

즉 connect와 rio_written은
return을 바로 함.
그러나 양쪽의 rio_readlineb는 block

Why Does Second Client Block



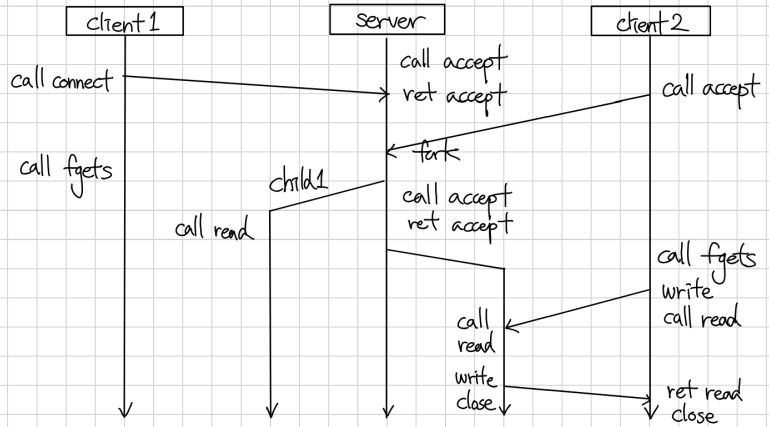
Approaches for Concurrent Servers

1) Process-based
- 서버가 fork를 이용, 여러 독립 프로세스 띄움
- Kernel이 automatically multiple logic flows 운영
- 각 flow는 각각의 private address space

2) Event-Based
- 하나의 프로세스 에서 일련의 이벤트처리 루틴 이용
- Programmer가 manually multiple logical flows 운영
- 모든 flows가 same address space 공유
- I/O multiplexing 사용

3) Thread-based
- 하나의 프로세스 에서 여러 thread 만듦
- Kernel이 automatically multiple logical flows 운영
- 각 flow가 same address space 공유
- process based와 event based의 혼합.

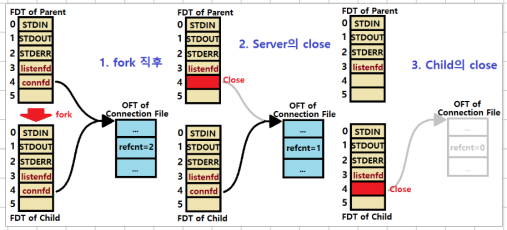
Approach #1: Process-based Servers



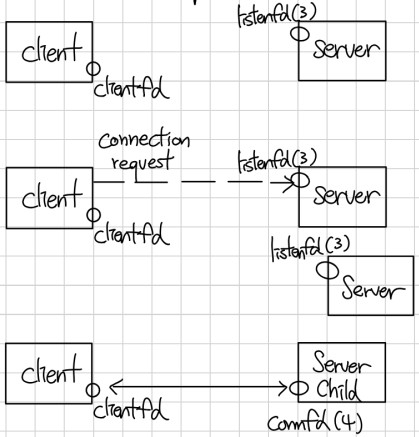

```c
int main(int argc, char **argv)
{
  int listenfd, connfd;
  socklen_t clientlen;
  struct sockaddr_storage clientaddr;

  Signal(SIGCHLD, sigchld_handler);
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    if (Fork() == 0) {
      Close(listenfd); /* Child closes its listening socket */
      echo(connfd);    /* Child services client */
      Close(connfd);   /* Child closes connection with client */
      exit(0);         /* Child exits */
    }
    Close(connfd); /* Parent closes connected socket (important!) */
  }
}
```

```c
void sigchld_handler(int sig)
{
  while (waitpid(-1, 0, WNOHANG) > 0)
    ;
  return;
}
```

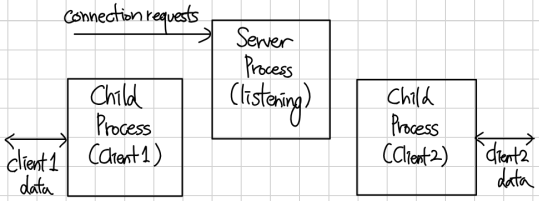


Concurrent Server : accept illustrated



1. server는 accept에서 block된 상태.
   listenfd에 request가 올때까지 wait
2. client에서 connect 함수를 호출하여 connection request를 보냄.
3. server는 accept를 통해 connfd를 return, connection 완성!!

Process-based Server Execution Model



- 각 client는 child process가 처리.
- 각 child 끼리 어떤 상태, 공간도 공유 X
- Parent는 connfd를 close
  Child는 listenfd를 close

Issues with Process-based Servers

- Listening server process는 zombie children을 반드시 reaping 해야함.
- Parent process는 반드시 connfd를 닫아야함.
  fork 이후 refcnt(connfd)=2
  connection은 refcnt(connfd)=0 이 되기 전까지 닫히지 X

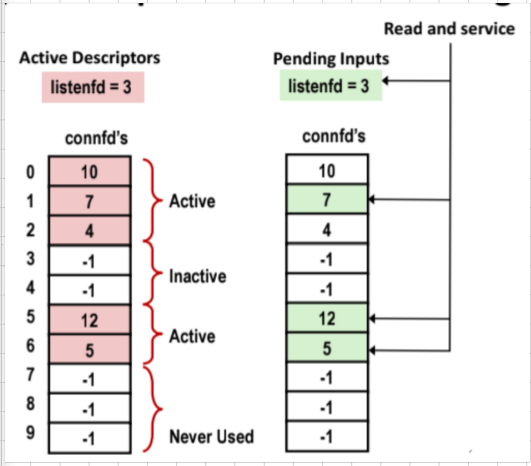
Pros and Cons of Event-based Servers

⊕ handle multiple connections concurrently
⊕ clean sharing model
  → descriptors (no, 비공유) 왜? 바로 close
  → file tables (yes)
  → global variables (no)
⊕ simple & straightforward

⊖ additional overhead for process control
⊖ nontrivial to share data between processes
  → requires IPC

Approach #2 Event-based Concurrent Server

→ server가 'Active Connection'에 대한 관리 맡음.
  ↳ 'Array of connfd's'
→ 어떤 descriptor에 pending input이 있는지 조사.
→ 만약 listenfd에 pending input ⇒ connection accept 수행 (array에 새 connfd 추가)
  connfd에 " ⇒ 서비스 처리



I/O Multiplexing-using select

int select(int n, fd_set *readfd, fd_set *writefd, fd_set *exceptfd, struct timeval *timeout)
⇒ active(ready) descriptor의 개수를 반환, 없으면 -1

void FD_ZERO(fd_set *fdset)	bit vector	0으로 초기화
void FD_CLR(int fd, fd_set *fdset)	fd에 위치한	k번째 bit 0으로.
void FD_SET(int fd, fd_set *fdset)	"	k번째 bit setting
int FD_ISSET(int fd, fd_set *fdset)		k번째 bit가 set인지 확인

ex) stdin(0번)에 1로 set, listenfd(3번)에 1로 set하면, 0과 3에 pending input이 있는지 확인 가능.

```c
#include "csapp.h"
void echo(int connfd);
void command(void);

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);

    FD_ZERO(&read_set);              /* Clear read set */
    FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
    FD_SET(listenfd, &read_set);     /* Add listenfd to read set */

```

※감시목록.

line 18:

	listenfd			stdin
	3	2	1	0
read_set (#):	0	0	0	0

line 19-20:

	listenfd			stdin
	3	2	1	0
read_set ({0,3}):	1	0	0	1

커널 타이핑 처리.

→ 모두 false로 초기화

→ stdin 활성화

→ listenfd 활성화

t) ready_set과 read_set 따로 두는 이유

read_set은 원본 감시 목록을 계속 유지해야함.

근데 select는 호출하면, fd_set을 변경 ⇒ 이벤트 발생한 FD만 1로 남기고, 나머지 0으로 만듦.

```c
he ready set
    while (1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
            command(); /* Read command line from stdin */
        if (FD_ISSET(listenfd, &ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            echo(connfd); /* Echo client input until EOF */
            Close(connfd);
        }
    }
}

void command(void) {
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    printf("%s", buf); /* Process the input command */
}
```

← 보사.

line 24:

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

→ stdin에서 이벤트

→ listenfd에서 상황발생.

Issues with select.c

```c
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* High water index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

int byte_cnt = 0; /* Counts total bytes received by server */

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);

    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready */
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

block이 안된다? → 요청이 온 녀석들만 해서?

```c
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

Pool of connected descriptors

```c
struct {
  int maxfd;
  fd_set read_set;
  fd_set ready_set;
  int nready;
  int maxi;
  int clientfd[FD_SETSIZE];
  rio_t clientrio[FD_SETSIZE];
} pool;
```

```c
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool high water mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

code/conc/echoservers.c

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

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                       n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }

            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

Pros and Cons of Event-based Servers

- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- **– Significantly more complex to code than process- or thread-based designs.**
- ~~**– Hard to provide fine-grained concurrency**~~
  - E.g., how to deal with partial HTTP request headers
- **– Cannot take advantage of multi-core**
  - Single thread of control