Unix IPC Overview

(Inter-Process Communication)

조영일

목차

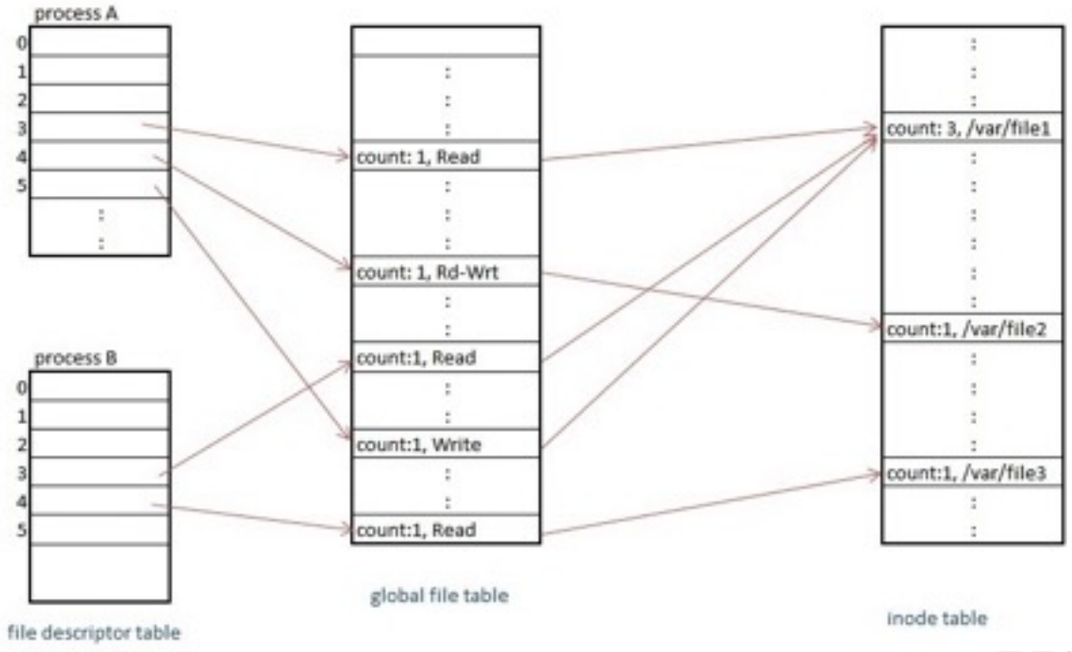
소개

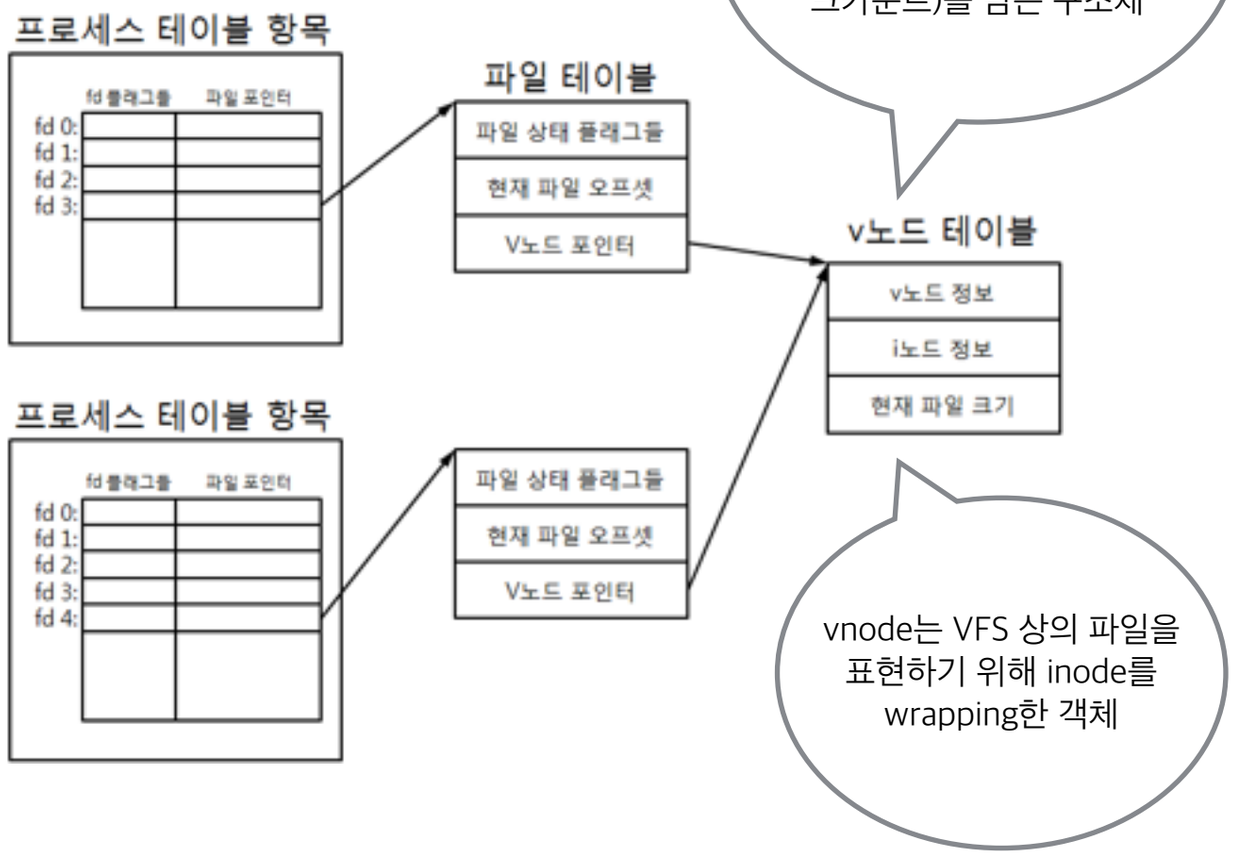
- IPC는 Inter-Process Communication의 약자
  - 프로세스끼리 정보를 전달하고 공유하는 여러 가지 방법을 총칭함
    - OS에서 제공하는 primitive를 이용해야 함
- 전통적인 IPC
  - Pipe, Semaphore, Shared Memory, Memory-Mapped File, Signal, …
- IPC ???
  - File, Socket, Message Passing

File

- 가장 단순한 방법
- 그러나, 가장 성능을 손해보는 방법
  - 메모리에 비해 보통 수십 배, 심하게는 수천 배까지 느림
- 방법
  - 하나의 프로세스가 파일에 write하고
  - 다른 프로세스가 파일에서 read하면 됨
  - 경우에 따라 file locking이 필요함
- JVM의 경우, VM 위에 프로세스가 단 한 개인 모델을 사용하므로 프로세스 간 통신은 파일에 의존하기도 함

File


process A
0
1
2
3
4
5
process B
0
1
2
3
4
5
file descriptor table
count: 1, Read
count: 1, Rd-Wrt
count:1, Read
count:1, Write
count:1, Read
global file table
count: 3, /var/file1
count:1, /var/file2
count:1, /var/file3
inode table


inode는 전통적인 유닉스 파일이 가지는 정보(크기, 디스크장치ID, 소유자, 그룹, 권한, timestamp, 링크카운트)를 담는 구조체
프로세스 테이블 항목
fd 플래그들
파일 포인터
fd 0:
fd 1:
fd 2:
fd 3:
파일 테이블
파일 상태 플래그들
현재 파일 오프셋
V노드 포인터
v노드 테이블
v노드 정보
i노드 정보
현재 파일 크기
프로세스 테이블 항목
fd 플래그들
파일 포인터
fd 0:
fd 1:
fd 2:
fd 3:
fd 4:
파일 상태 플래그들
현재 파일 오프셋
V노드 포인터
vnode는 VFS 상의 파일을 표현하기 위해 inode를 wrapping한 객체

Socket

- IP 주소와 Port 번호를 가지는 프로세스끼리 통신하는 인터페이스
  - 주로 TCP/UDP 통신에 사용되는 인터페이스
- 프로토콜은 애플리케이션 수준에서 정해야 함
  - HTTP, FTP는 이미 정해진 프로토콜
  - /etc/services: IANA에서 포트 번호 별로 어떤 프로토콜을 사용할지 정해놓은 파일 (1024까지는 표준 포트 번호)
- 장점
  - 네트웍 연결을 파일처럼 사용할 수 있음 (socket descriptor)
- 단점
  - TCP의 connection setup 비용이 상당히 크기 때문에 전통적인 IPC에 비해 과함
    - 파이프로 대체할 것

* 프로토콜

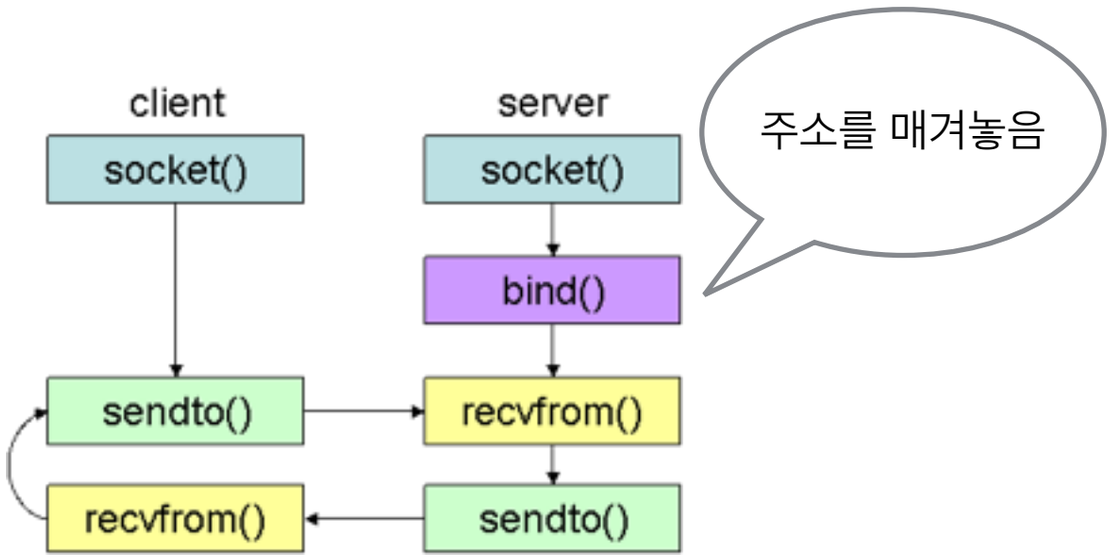
- Ex) 메시지를 TLV 포맷으로 주고받자!
  - “Q9HOWAREYOU”
  - “A6IMFINE”

- HTTP Request
  - GET http://www.naver.com HTTP/1.1

- HTTP Response

```
HTTP/1.1 200 OK
Date: Fri, 25 Nov 2011 05:47:42 GMT
Server: Apache
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
```

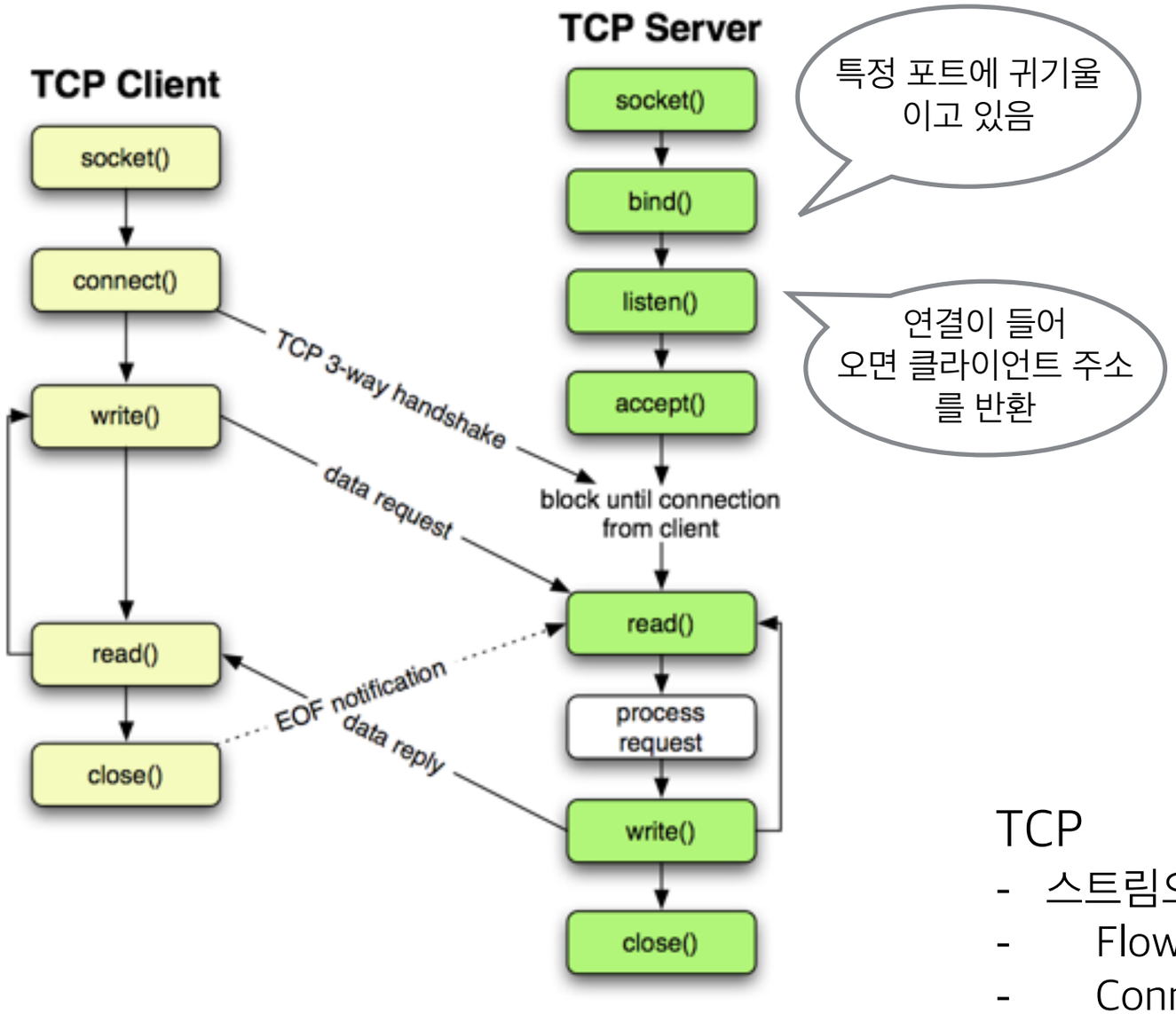
Socket



UDP

- 패킷 단위의 전송
- Flow control, congestion control이 지원되지 않으므로, 필요하다면 애플리케이션 수준에서 그런 기능을 구현해야 하는 부담
- 가벼움

Socket



TCP
- 스트림으로 전송
- Flow control, congestion control이 지원됨
- Connection setup/teardown이 복잡함

Unix Domain Socket

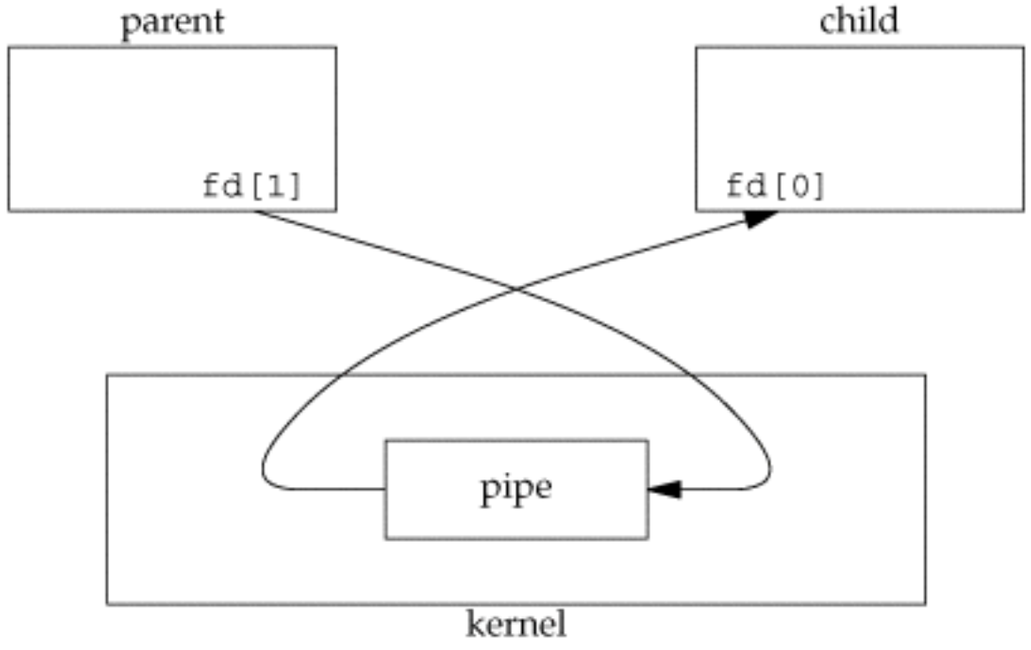
- Socket API를 사용하지만 IP 주소를 가지지 않음
  - 같은 서버 내의 프로세스끼리 통신할 때 편리함
- IP 주소와 Port 번호 대신 파일을 공유함
  - 그러나 파일에 데이터가 쌓이지 않고 커널이 중계해줌

```
socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(IPADDR_ANY);
Serveraddr.sin_port = htons(port_num);
```

```
socket(AF_UNIX, SOCK_STREAM, 0);
serveraddr.sun_family = AF_UNIX;
strcpy(serveraddr.sun_path, file_path);
```

Pipe

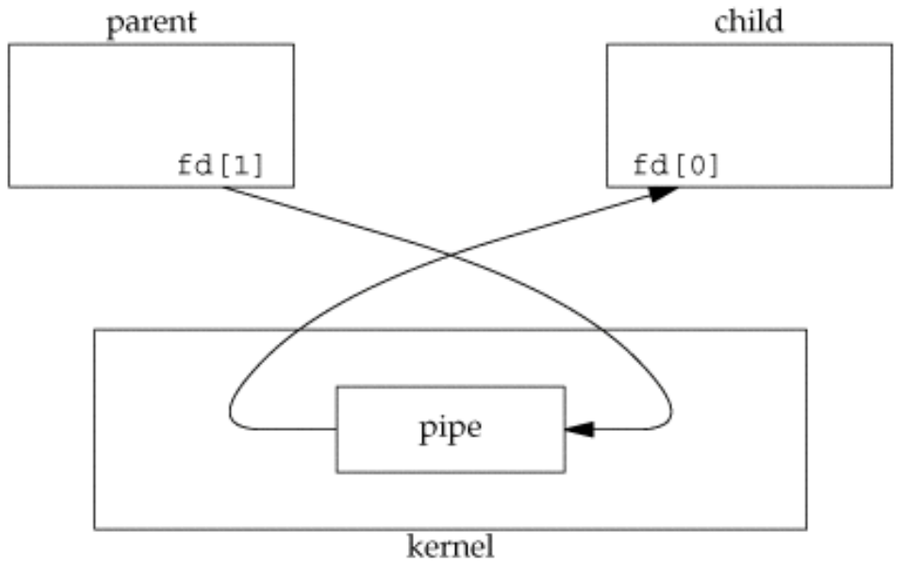
- 부모 프로세스와 자식 프로세스 간 공유하는 descriptor 형태의 연결
  - FIFO
  - 기본적으로는 one-way



Pipe

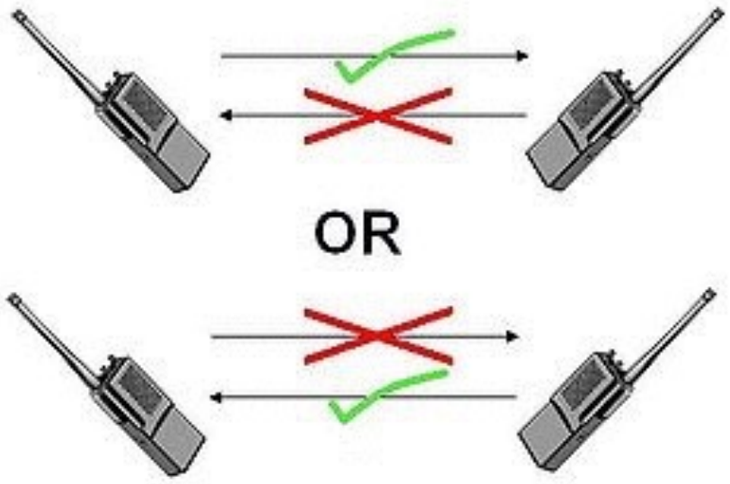
- 사용방법

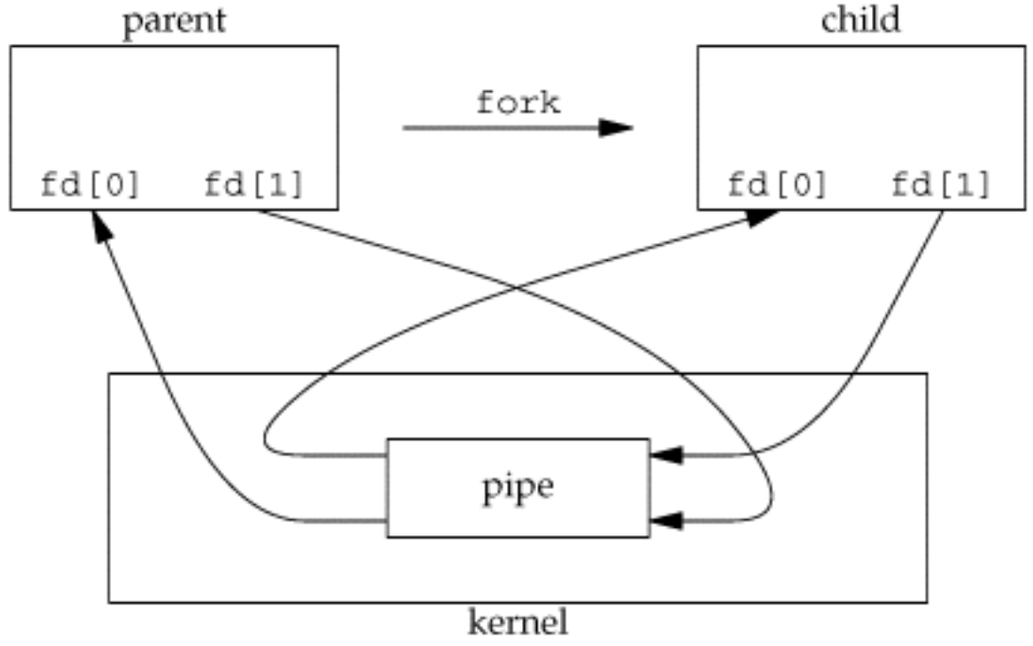
```
int pds[2] = pipe();
if (fork() > 0) {
    // parent (writer)
    close(pd[0]);
    write(pd[1], buf, buf_len);
} else {
    // child (reader)
    close(pd[1]);
    read(pd[0], &buf, 1024);
}
```



Pipe

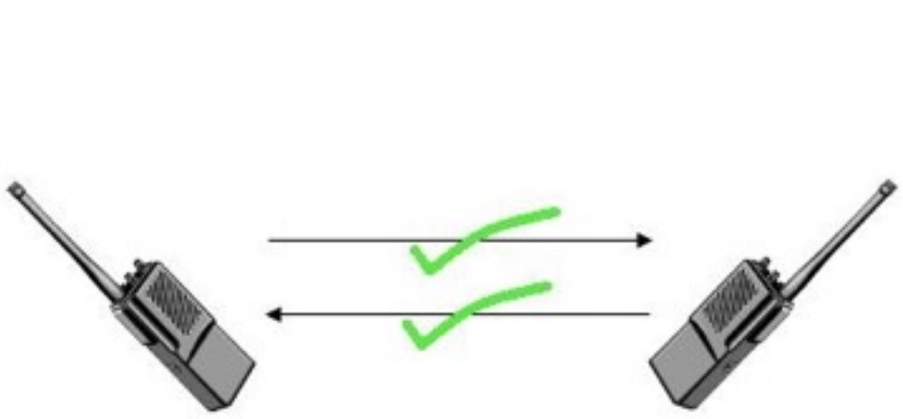
- Half-duplex
  - Pipe 하나를 read descriptor와 write descriptor로 병용
  - 그러나 한 번에 한 방향으로만 보낼 수 있다. (half-duplex)

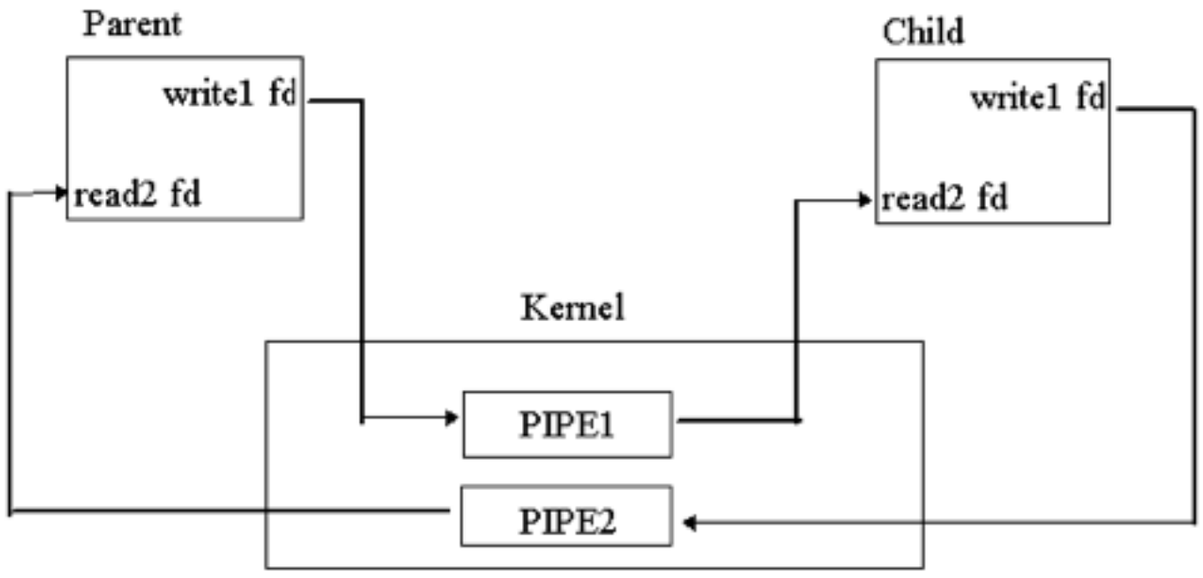




Pipe

- Full-duplex로 만들기
  - Pipe를 2개 만들어서 서로 반대방향으로 흐르게 하면 됨





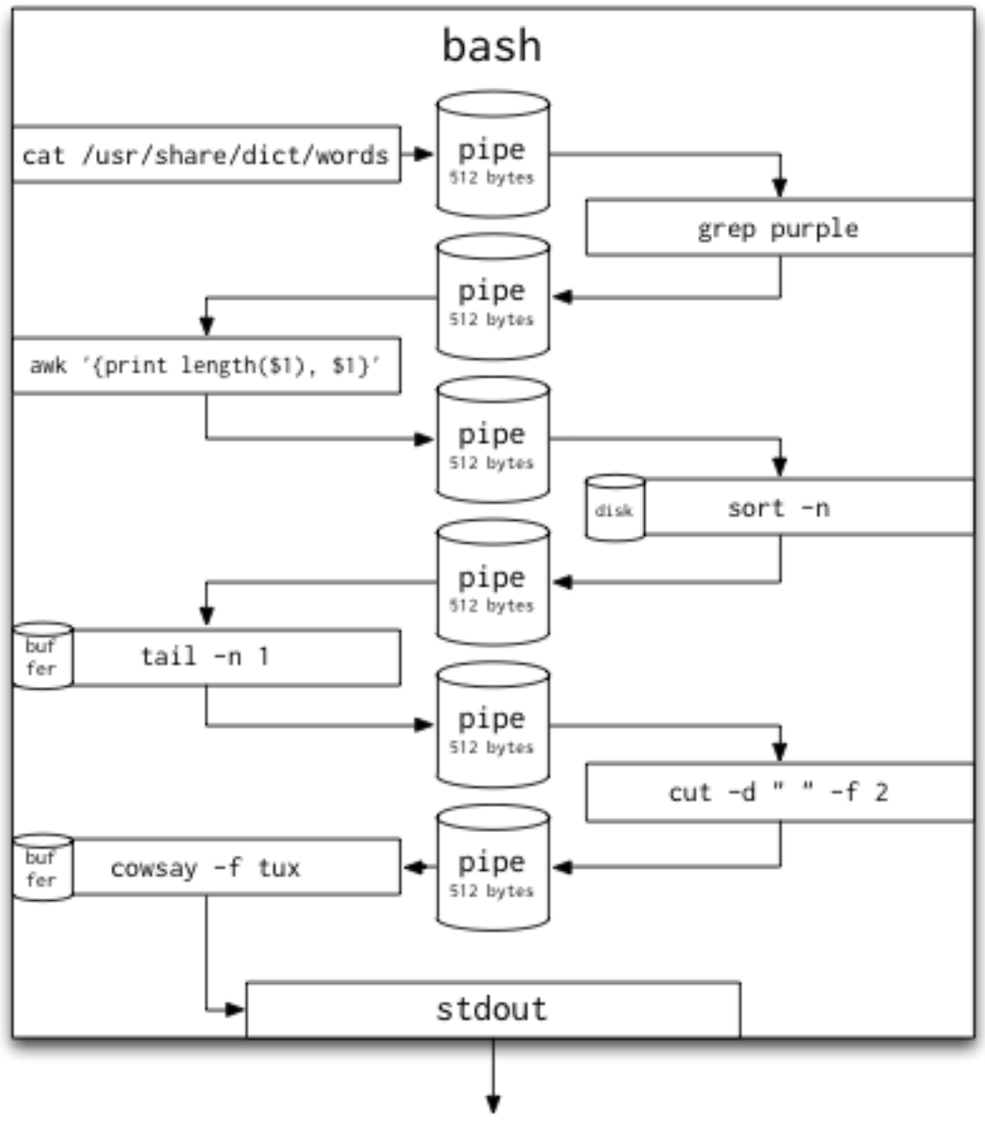
Pipe

- pipe() 함수 대신 popen()을 이용한 다른 용법
  - 현재 프로세스가 shell과 통신할 때 사용
    - Read-only
    - Write-only

```
popen(“ls”, “r”);

popen(“grep hello”, “w”);
```

Pipe

- 용도
  - Parent와 child process 간 통신
  - Unix shell의 filter
    - cat | grep | awk | sort | tail | …



Named Pipe

- “Named”는 파일로 존재한다는 의미
  - 일반 pipe가 부모-자식 관계여야만 사용할 수 있는데 반해 named pipe는 임의의 프로세스가 (파일을 통해) 연결 가능함
  - aka FIFO

- 사용방법

```
mknod(“mypipe”, SIFIFO | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP, 0);
or
mkfifo(“mypipe”, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);

int fd = open(“mypipe”, O_WRONLY);

int fd = open(“mypipe”, O_RDONLY);
```

Message Queue

- 특징
  - 커널에서 전역적으로 관리함
    - 어떤 프로세스든 접근이 가능함
  - 사용이 간단함
    - 소켓이나 파이프와 달리 receiver가 미리 대기하고 있을 필요가 없음

- API
  - System V message queue
    - msgget(), msgsnd(), msgrcv(), …
  - POSIX message queue
    - mq_open(), mq_send(), mq_receive(), …
    - mq_timedsend(), mq_timesreceive()

Message Queue

- 사용방법

```
char msg[] = “hello”;
unsigned int priority = 1;
mqd_t mqd = mq_open(“test_queue”, O_WRONLY | O_CREAT);
mq_send(mqd, msg, strlen(msg) + 1, priority);
```

```
char msg[1024];
unsigned int priority = 0;
mqd_t mqd = mq_open(“test_queue”, O_RDONLY);
mq_receive(mqd, msg, 100, &priority);
```

- 주의 사항
  - 큐가 가득 차거나 비어 있으면 함수 호출이 블럭됨
    - timed 함수를 이용하거나
    - O_NONBLOCK 플래그를 사용

Signal

- 용도
  - 기본적으로 Unix의 시그널은 특정 프로세스를 죽이라는 신호
  - 그러나 살해 명령(signal number)도 다양하고 receiver의 반응(action)도 다양함

type	num	defautl action	comment
SIGHUP	1	Term	프로세스 그룹의 리더가 죽었을 때 발생
SIGINT	2	Term	Ctrl-C
SIGKILL	9	Term	Kill (무시할 수 없음)
SIGSEGV	11	Core	segmentation fault
SIGTERM	15	Term	Graceful termination
SIGUSR1 SIGUSR2		Term	사용자 정의 시그널
SIGCHLD		Ign	자식 프로세스가 종료되었음
SIGSTOP		Stop	프로세스를 중단 (무시할 수 없음)
SIGCONT		Cont	중단된 프로세스를 다시 시작

Signal

- API
  - kill(pid, sig_num)
  - signal(sig_num, handler)

- 사용방법

```
void handler(int signum)
{
    printf(“signal %d” is catched”, signum);
}
if (signal(SIGHUP, handler) == SIG_ERROR) {
    perror(“can’t register a signal”);
    return -1;
}
_____________________________________________

kill(345, SIGHUP);
```

Signal

- 문제점
  - 같은 시그널에 대해 한 번에 하나씩만 블럭됨
  - 핸들러 수행 중 다른 시그널이 발생하면 처리가 중단됨

- 대안

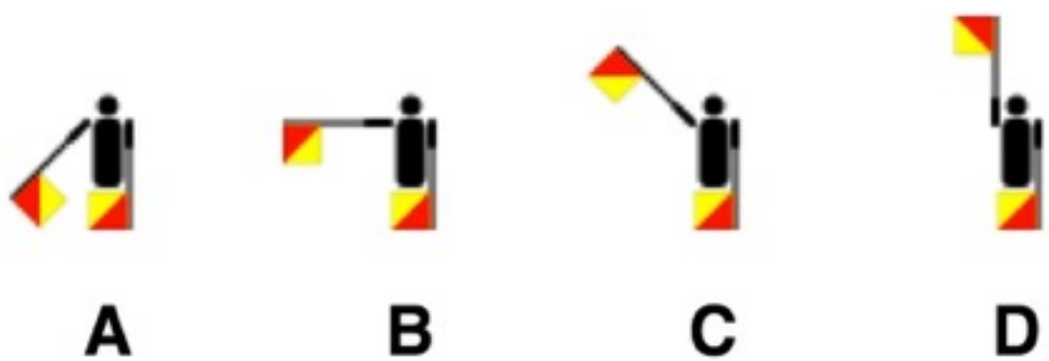
```
void sig_int(int sig_num) {
    sigset_t sigset, oldset;
    sigfillset(&sigset); // 모든 신규 시그널을 블럭
    sigprocmask(SIG_BLOCK, &sigset, &oldset);
}

struct sigaction intsig;
intsig.sa_handler = sig_int;
sigemptyset(&intsig.sa_mask); // 어떤 시그널에도 연결되지 않음
intsig.sa_flags = 0;
sigaction(SIGINT, &ingsig, 0); // INT 시그널에 등록
```

Semaphore

- Synchronization primitive
  - 프로세스와 쓰레드

- 개념
  - 깃발이 올려져 있으면 진행해도 되고
  - 깃발이 내려져 있으면 진행할 수 없음

- 동시에 사용할 수 있는 자원의 개수에 따라
  - Binary semaphore
  - Counting Semaphore



Semaphore

- 연산
  - P: test(decrement)
  - V: increment

- API
  - System V semaphore
    - semget(), semctl(), semop()
  - POSIX semaphore
    - /dev/shm에 sem.<이름>의 파일로 생성됨
    - for unnamed semaphore
      - sem_init(), sem_destroy()
    - for named semaphore
      - sem_open(), sem_close()
    - sem_wait(), sem_post()

Semaphore

```
char sem_name[] = “/mysem”;
sem_unlink(sem_name);
sem_t *mysem = sem_open(sem_name, O_CREAT, 0664, 1);
while (1) {
    sem_wait(mysem);
    // critical section
    sem_post(mysem);
}
sem_close(mysem);
```

```
sem_t *mysem = sem_open(sem_name, 0, 0664, 0);
while (1) {
    sem_wait(mysem);
    // critical section
    sem_post(mysem);
}
sem_close(mysem);
```

Semaphore

- 문제점
  - Starvation(기아)
    - 특정 프로세스가 계속 자원을 사용하지 못한 채로 대기할 가능성이 있음
    - 큐잉을 통해 순서를 정해주면 해결됨
  - Deadlock(교착)

프로세스 A	프로세스 B
P(S)	P(Q)
P(Q)	P(S)

Mutex

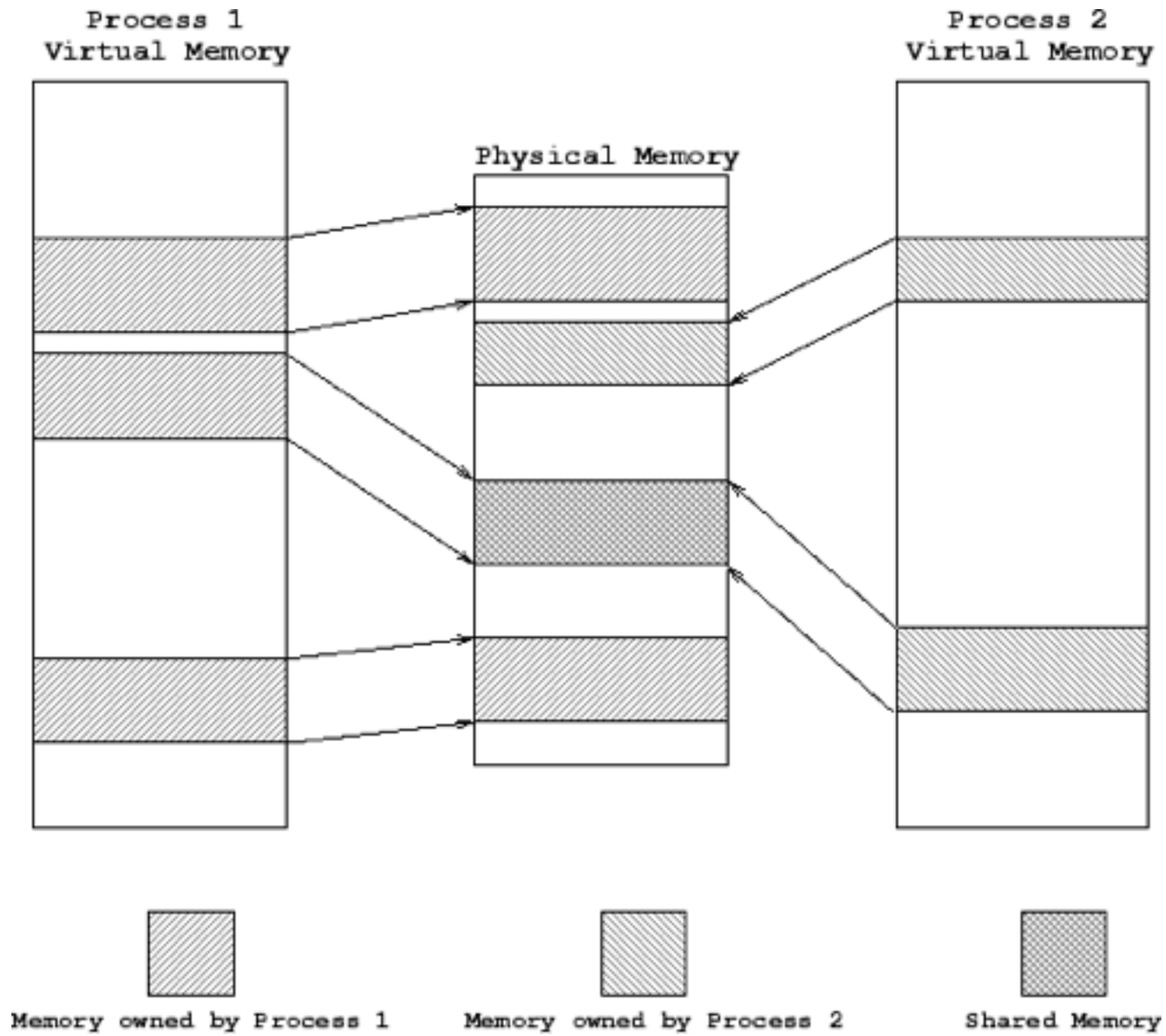
- 상호 배제(Mutual Exclusion)
  - 임계 영역(critical section)을 한 순간에 한 프로세스나 쓰레드만이 접근하도록 하는 장치
    - 임계 영역: 하나 이상의 쓰레드가 접근하면 안 되는 공유 자원을 사용하는 코드 조각

- API
  - pthread_mutex_init(), pthread_mutex_destroy()
  - pthread_mutex_lock(), pthread_mutex_unlock()

- Semaphore와의 차이
  - semaphore는 자원에 대한 동시 접근을 막는 도구
  - mutex는 코드에 대한 동시 접근을 막는 도구 (소유권 개념)

Shared Memory

- 프로세스가 사용할 수 있는 메모리의 종류
  - Private memory
  - Shared memory



Shared Memory

- API
  - System V
    - shmget(), shmat(), shmdt(), shmctl()
  - POSIX
    - shm_open(), shm_unlink()

- 문제점
  - detach는 따로 ipcrm를 실행해줘야 완전히 반환됨
  - 반면, unlink는 접근하는 프로세스가 모두 사라지면 그때 반환됨

Shared Memory

- 사용방법

```
struct region { int len, char buf[MAX_LEN]; };
unsigned int size = sizeof (struct region);
char shm_name[] = “/myshm”;

shm_unlink(shm_name);
int fd = shm_open(shm_name, O_CREAT | O_RDWR, 0777);
ftruncate(fd, size);
struct region *ptr =
     mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
munmap(ptr, size);
close(fd);
```

Memory Mapped File

- 파일을 메모리처럼 다룰 수 있는 사용방식
  - seek 대신 포인터로 특정 주소에 접근가능
  - 자료구조를 바로 읽어내거나 쓸 수 있음

- 다양한 매핑 방식
  - anonymous: 스왑메모리에 IO
  - file-backed: 파일에 직접 IO
  - ...

- 용도
  - OS가 프로세스를 load할 때
  - 프로세스 간 메모리 공유 시
  - ...

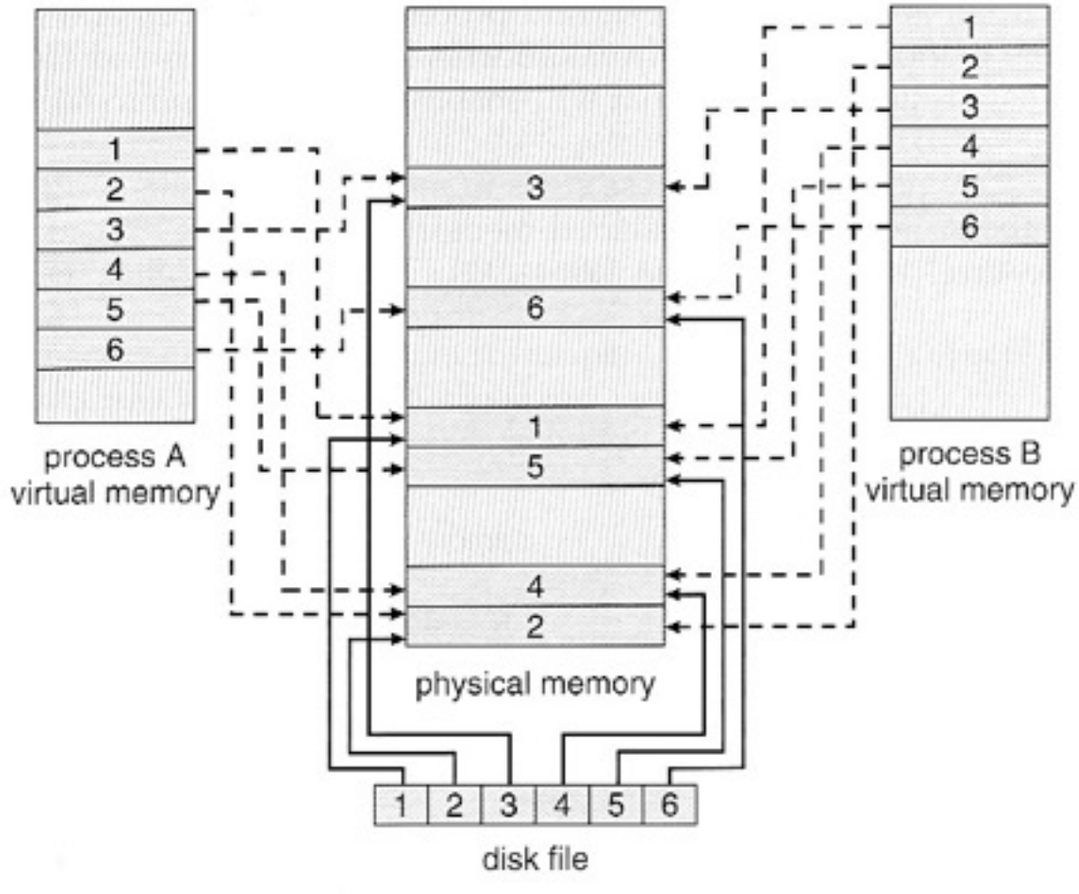


Figure 9.23 Memory-mapped files.

Memory Mapped File

- API
  - mmap(), munmap(), msync()
- 주의사항
  - msync()를 강제로 호출해줘야 프로세스 간 동기화됨
- Shared memory와의 비교
  - shm: persistent
  - mmap: file-backed 매핑이 아니면 프로세스 종료 시 사라짐

Memory Mapped File

- mmap 사용방법

```
struct stat sb;
int fd = open(filename, O_RDONLY);
fstat(fd, &sb);
char* addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
write(STDOUT_FILENO, addr, sb.st_size);
```

Message Passing

- 통신 방법의 한 가지 형태
  - 문자열이나 자료구조, 코드를 다른 프로세스에 전달
  - 표준 API가 존재하지 않음
    - 구현체가 다양함
      - RPC, RMI, Corba, SOAP, DCOM, ...

IO Multiplexing

- Challenges
  - IO는 기본적으로 blocking이 발생한다.
    - read()를 호출했는데 해당 자원(파일, 소켓, 파이프, ...)에 새로 write된 내용이 없으면 이 함수는 반환되지 않음
    - vice versa
  - 하나의 프로세스가 여러 자원으로부터 데이터를 읽거나 쓰려고 한다면?
    - 소켓과 파이프에서 데이터를 읽어서 파일에 데이터를 쓰는 경우
    - 여러 자식 프로세스와 각각 파이프로 연결해 놓은 경우
    - 소켓과 파이프에 대해 read()를 동시에 호출할 수 없음
      - 한 쪽의 read()가 반환될 때까지 기다려야 함
  - IO Multiplexing이란 여러 자원에 대해 읽고 쓸 수 있는 방법

IO Multiplexing

- 무식한 방법
  - Polling
    - IO 준비가 되었는지 모든 자원(fd)를 검사해보고 안 되었으면 기다림
- 덜 무식한 방법
  - locking with multi-process/threads
    - 어려움
- 교양있는 방법
  - Select/Poll
- 최신 유행
  - Epoll/EventFD

IO Multiplexing

- Select
  - 기본적으로 1024개 이하의 자원을 비트 테이블로 관리
  - select가 대표로서 blocking mode로 IO 준비가 되기를 기다림
  - IO 준비가 되면 각각의 fd를 검사하여 필요한 작업을 하면 됨
  - read/write/error로 fd를 분류해놓음

IO Multiplexing

- Select 사용방법

```
int sockfd, pipefd;
fd_set rfds, wfds;
FD_ZERO(&rfds); FD_ZERO(&wfds);
FD_SET(sockfd, &rfds); FD_SET(pipefd, &wfds);
while (1) {
    restart:
    struct timeval timeout = { 5, 0 };
    if (select(2, &rfds, &wfds, NULL, &timeout) < 0) {
        if (errno == EINTR) {
            goto restart;
        }
    }
    if (FD_ISSET(sockfd, &rfds)) {
        // accept ...
    }
    if (FD_ISSET(pipefd, &wfds)) {
        // write...
    }
}
```

IO Multiplexing

- Poll
  - 1024개 이상의 자원을 배열로 관리
  - 자원마다 관심있는 이벤트를 등록해 놓음

```
struct pollfd fds[2];
fds[0].fd = open("/dev/dev0", ...);
fds[1].fd = open("/dev/dev1", ...);
fds[0].events = POLLOUT | POLLPRI;
fds[1].events = POLLOUT | POLLWRBAND;
while (1) {
    ret = poll(fds, 2, timeout_msecs);
    if (ret < 0) { return -1; }
    for (i = 0; i < 2; i++) {
        if (fds[i].revents & POLLHUP || fds[i].revents & POLLERR || fds[i].revents & POLLINVAL) { return 0; }
        if (fds[i].revents & POLLOUT) { handle_normal_data(); }
        if (fds[i].revents & POLLPRI) { handle_hi_priority_data(); }
    ...
```

IO Multiplexing

- Event-driven IO
  - Select나 Poll을 사용한다고 하더라도 실제 처리는 동시에 진행되는 게 아니라 해당 fd에 대해서만 처리하는 게 아닌가?
  - 예제 수준으로 만들면 blocking IO와 별반 성능 차이 없음
  - 실제의 대규모 처리 서버에서는 별도의 프로세스나 쓰레드를 만들어서 처리함
  - 그러나 100개만 넘어가도 context-switching 비용이 크게 증가함
- 결국 event-driven IO를 도입해야 response time을 높일 수 있음
  - 부가적으로 독자적인 스케쥴링 알고리즘을 채택할 수 있음

IO Multiplexing

- Epoll
  - event-driven 방식
  - “Free lunch”

- API
  - epoll_create()
  - epoll_ctl
  - epoll_wait()

IO Multiplexing

- Epoll 사용방법

```
#define MAX_EVENTS 10000
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = sockfd;
struct epoll_event all_evs[MAX_EVENTS];
int epfd = epoll_create(MAX_EVENTS);
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &ev);
while (1) {
    int nfds = epoll_wait(epfd, &all_evs, MAX_EVENTS, 100 /*ms*/);
    if (nfds == 0) continue;
    if (nfds < 0) break; // wait error
    for (i = 0; i < nfds; i++) {
        int fd = all_evs[i].data.fd;
        // process the IO of fd
    }
}
```

IO Multiplexing

- Epoll
  - Event Distribution
    - Level-trigger: 데이터가 존재하면 이벤트를 배달해 줌
      - Select/Poll-style
    - Edge-trigger: 감시 중인 fd들에서 이벤트가 발생했을 때만 이벤트를 배달해 줌
    - Ex) writer가 2KB를 write했는데 reader가 1KB만 read한 상황
      - LT라면 데이터가 남았으니 이벤트를 계속 배달해 주고
      - ET라면 남은 데이터와 상관없이 이벤트를 배달해 주지 않음
    - ET에서는 hang이 발생할 수 있으므로 반드시 non-blocking IO로 구현해야 함

IO Multiplexing

- Epoll
  - Epoll도 non-blocking IO 방식으로 바꿔야 CPU utilization을 높일 수 있음
  - Non-blocking IO

```
fcntl(fd, F_SETFL, flags |
O_NONBLOCK);
restart:
while (1) {
    nread = read(fd, buf, size);
    // process the buffer data
    if (nread < 0 && errno == EAGAIN)
        goto restart;
    // process the buffer data
}
```

IO Multiplexing

- EventFD
  - Full-duplex, pipe descriptor-style semaphore
    - 파이프처럼 프로세스를 연결하는 디스크립터
    - 64비트 정수값 하나만을 주고받는 파이프임
    - 세마포어처럼 0보다 큰 값이 write되면 다른 프로세스로 보내지고 0이 되면 read가 block됨
  - introduced in kernel 2.6.22, revised in 2.6.27 & 30
  - harmonized with select()/poll()

- API
  - eventfd(unsigned int initval, int flags)

IO Multiplexing

- EventFD 사용방법

```
uint64_t u = 0;
int efd = eventfd(0, 0);
if (fork() == 0) {
    // child
    u = 3;
    sleep(2);
    write(efd, &u, sizeof (uint64_t));
} else {
    // parent
    read(efd, &u, sizeof (uint64_t));
}
```


THANKS
THAT'S ALL
DIYLOL.COM