

목 차

제 1 장 네트워크 프로그래밍 개요	1
1.1 컴퓨터 통신 프로토콜	3
1.1.1 컴퓨터 통신 프로토콜의 정의	3
1.1.2 OSI 7 Layer	4
1.1.3 TCP/IP	6
1.1.4 통신 프로토콜 표현 방식	9
1.2 네트워크 프로그래밍의 분류	11
1.3 클라이언트-서버 모델	13
1.3.1 클라이언트-서버 모델의 정의	13
1.3.2 서버 구현 기술	18
제 2 장 TCP/IP 프로토콜	20
2.1 네트워크 액세스 계층	22
2.1.1 이더넷	22
2.1.2 PPP	25
2.1.3 기타 서브네트워크	26
2.2 IP 프로토콜	27
2.2.1 IP 데이터그램 구조	28
2.2.2 IP 계층 기능	35
2.3 TCP 프로토콜	38
2.3.1 TCP 특징	38
2.3.2 TCP 헤더	42
2.3.3 TCP 연결설정	46
2.3.4 TCP 연결종료	47
2.3.5 데이터 송수신	51
2.4 UDP 프로토콜	56
제 3 장 소켓 프로그래밍	59
3.1 소켓의 이해	61
3.1.1 소켓 정의	61
3.1.2 소켓 사용법	65
3.2 인터넷 주소변환 체계	74
3.2.1 바이트 순서	74
3.2.2 IP 주소변환	77
3.3 TCP 기반 프로그램	84
3.3.1 TCP 클라이언트 프로그램	84
3.3.2 TCP 클라이언트 예제 프로그램	89
3.3.3 TCP 서버 프로그램	95
3.3.4 TCP 에코 서버 프로그램	100
3.4 UDP 프로그램	104

3.4.1 UDP 프로그램 작성 절차	104
3.4.2 UDP 에코 프로그램	106
제 4 장 고급 소켓 프로그래밍	114
4.1 소켓의 동작 모드	116
4.2 다중처리 기술	118
4.2.1 멀티태스킹	118
4.2.2 다중화	120
4.3 비동기형 채팅 프로그램	122
4.3.1 채팅 서버 프로그램 구조	122
4.3.2 select()	124
4.3.3 채팅 서버 프로그램	127
4.3.4 채팅 클라이언트 프로그램	134
4.4 폴링형 채팅 프로그램	138
4.4.1 fcntl()	138
4.4.2 폴링형 채팅 서버	141
제 5 장 소켓 옵션	150
5.1 소켓 옵션 종류	152
5.1.1 SO_KEEPALIVE	152
5.1.2 SO_LINGER	153
5.1.3 SO_RCVBUF와 SO_SNDBUF	156
5.1.4 SO_REUSEADDR	157
5.1.5 기타 옵션	159
5.2 소켓 옵션 변경	162
5.2.1 소켓 옵션 변경 함수	162
5.2.2 소켓 옵션 변경 예제 프로그램	164
제 6 장 프로세스	167
6.1 프로세스의 이해	169
6.1.1 프로세스	169
6.1.2 프로세스 식별자 확인	170
6.2 프로세스의 생성과 종료	174
6.2.1 프로세스의 생성	174
6.2.2 프로세스의 종료	175
6.3 데몬 서버 구축 방법	176
6.3.1 데몬 프로세스	176
6.3.2 데몬 서버 종류	178
제 7 장 시그널	182
7.1 시그널 종류	184
7.1.1 시그널의 종류	184
7.2 시그널 처리	186
7.2.1 시그널 처리 기본 동작	186

7.2.2	시그널 핸들러	187
7.2.3	시그널 처리 예	193
7.3	SIGCHLD와 프로세스의 종료	196
7.3.1	프로세스의 종료	196
7.3.2	프로세스의 종료 처리	199
제 8 장	프로세스간 통신	206
8.1	파이프	208
8.1.1	파이프 생성	208
8.1.2	파이프를 이용한 에코 서버 프로그램	211
8.2	FIFO	218
8.2.1	FIFO의 정의	218
8.2.2	FIFO를 이용한 에코 서버 프로그램	220
8.3	메시지큐	226
8.3.1	메시지큐 개요	226
8.3.2	메시지큐 생성	228
8.3.3	메시지 송수신	232
8.3.4	메시지큐 제어	239
8.3.5	메시지큐를 이용한 에코 서버	243
8.4	공유메모리	251
8.4.1	공유메모리 생성	252
8.4.2	공유메모리 제어	256
8.4.3	공유메모리의 동기화문제 처리	259
8.5	세마포어	264
8.5.1	세마포어 사용	264
8.5.2	세마포어 제어	268
8.5.3	세마포어 이용 예	273
8.5.4	공유메모리의 동기화문제 처리	277
제 9 장	스레드 프로그래밍	283
9.1	스레드의 생성과 종료	285
9.1.1	스레드 생성과 종료	286
9.1.2	스레드의 상태	294
9.2	스레드 동기화	296
9.2.1	동기화 문제	296
9.2.2	뮤텍스	300
9.2.3	뮤텍스 사용 예	306
9.3	스레드간 통신	309
9.3.1	조건변수 사용 방법	309
9.3.2	조건변수 사용 예	313
9.4	멀티스레드 에코 서버 프로그램	318

Linux Network & System Programming

1장

네트워크 프로그래밍 개요

- 1.1 컴퓨터 통신 프로토콜
- 1.2 네트워크 프로그래밍의 분류
- 1.3 클라이언트 - 서버 모델

Overview

- 네트워크에 대한 기본적인 지식 습득
- 서비스에 대한 종류와 서비스에 대한 기본적인 지식 습득
- 네트워크 프로그램을 설계하는 데 필요한 서버 구축 기법

1.1 컴퓨터 통신 프로토콜

1.1.1 컴퓨터 통신 프로토콜의 정의

통신 프로토콜의 특징

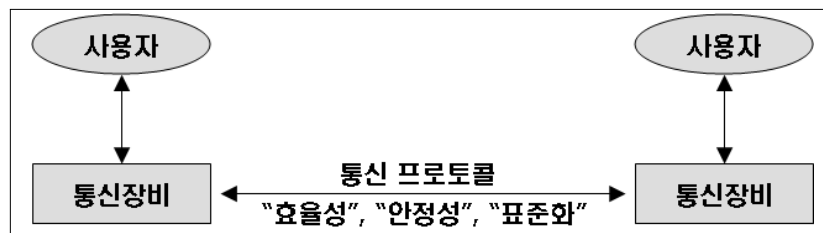


그림 1.1 통신 프로토콜의 특징

통신 프로토콜의 종류

표준 통신 프로토콜, 사용자 정의 통신 프로토콜

컴퓨터 통신 프로토콜의 정의

- 컴퓨터들이 데이터를 원활히 주고받을 수 있도록 정한 약속

통신 프로토콜의 특징

- 효율성 : 주어진 통신 채널을 최대한 이용할 수 있도록 프로토콜을 정함, 흐름제어
- 안정성 : 비정상적인 외부요인이 발생한 경우에도 통신은 안정되게 동작 또는 종료, 에러제어
- 표준화 : 널리 사용되기 위해 표준화
 - IETF(Internet Engineering Task Force)에 의해 RFC라는 문서로 제공
 - <http://ietf.org/rfc.html>

통신 프로토콜의 종류

- 표준 통신 프로토콜 : 국제적으로 통용되는 표준(TCP/IP, http, telnet, ftp, IEEE 802.3 등)
- 사용자 정의 통신 프로토콜 : 개발자가 특정 서비스를 제공하기 위해 임의로 정한 프로토콜

1.1.2 OSI 7 Layer



그림 1.2 OSI 7계층 프로토콜 구조와 TCP/IP 프로토콜 구조

물리 계층(physical layer)

- 정보의 최소 단위인 비트를 전송매체를 통하여 효율적으로 전송하는 기능을 정의

데이터 링크 계층(link layer)

- 물리 계층의 서비스를 이용하여 다수의 비트로 구성된 프레임(PDU : Protocol Data Unit)을 노드 사이에 신뢰성 있게 전송하는 기능 정의
- 링크의 개설과 해제, 프레임의 경계 식별, 에러제어, 흐름제어 등을 수행

네트워크 계층(network layer)

- 링크 계층에 의해 프레임에 실려온 패킷을 라우터 등의 교환 장치를 거쳐 목적지 호스트까지 전달하는 교환기능
- 주소의 분석, 호스트 사이의 논리적 연결설정과 해제, 패킷 단위의 흐름제어, 경로 배정
- 데이터의 직접적인 운반이 아닌 패킷의 교환 기능만 처리

전달 계층(transport layer)

- 네트워크 계층을 이용한 종점 호스트 사이의 데이터 송수신
- 종점간(end-to-end) 연결관리, 에러제어, 흐름제어
- 종점 호스트에서만 수행되는 종점간 프로토콜이며 라우터 등의 내부장비에서는 처리하지 않음

세션 계층(session layer)

- 전달 계층을 이용하여 통신 서비스의 개설, 서비스 유지 및 종료
- OSI하위 계층(1-4)의 기능을 이용하여 종점 호스트 프로세서에서 이루어지는 통신 관리 프로토콜

표현 계층(presentation layer)

- 데이터의 표현 방식이 서로 다른 호스트들 사이의 통신 지원
- 데이터의 표준화된 표현 방식의 사용
- 코드 변환, 데이터 압축, 데이터의 암호화·복호화 수행

응용 계층(application layer)

- 네트워크를이용한 최종 응용 서비스를 제공하는 계층
- ftp, telnet, e-mail, 웹서비스, 망관리, 분산처리와 같은 네트워크 응용 기능 처리

1.1.3 TCP/IP

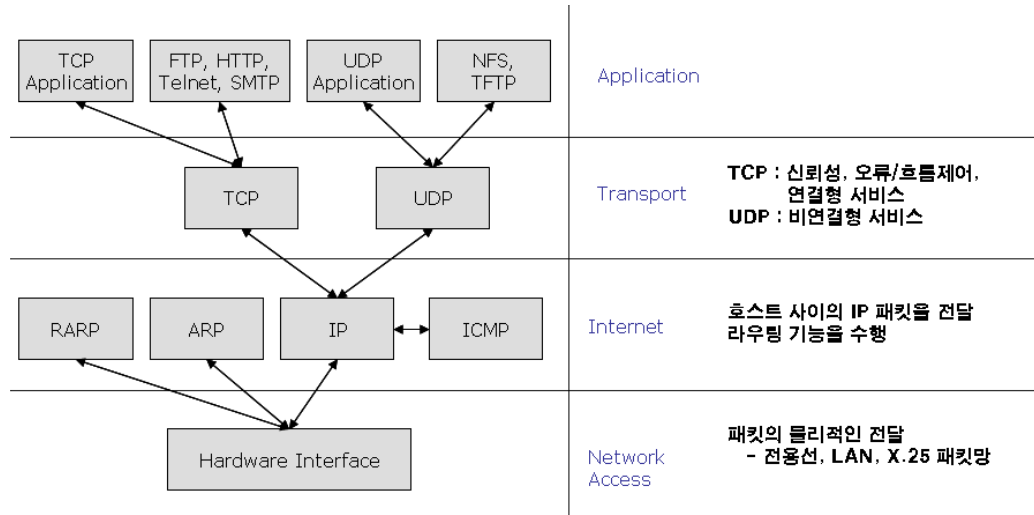


그림 1.3 TCP/IP 프로토콜 구조

네트워크 액세스 계층

- IP 데이터그램을 전달하기 위해 이더넷, DSL(Digital Subscriber Line)등의 서브네트워크 이용

인터넷 계층

- IP계층이라고도 하며 IP 데이터그램을 최종 ahrwjrwIRK지 전달하는 기능
- IP 데이터그램을 어떤 경로로 전달할지 정하는 라우팅 기능 수행
- ICMP
 - IP 데이터그램의 전송을 돕는 제어 프로토콜
 - 호스트 또는 라우터 사이에 에러정보나 제어정보 전달
- IGMP
 - 멀티캐스팅을 지원하는 프로토콜

트랜스포트 계층

TCP	UDP
<ul style="list-style-type: none"> - 신뢰성 있는 연결형 서비스 - ftp, telnet, e-mail, http 등에서 사용 - 신뢰성 있는 통신의 제공을 위해 확인응답(Ack), 체크섬(checksum), 재전송, 종점간 흐름제어 - 종점간 바이트 스트림의 제공 : 송신측에서 전송된 데이터가 바이트 단위로 차례대로 전달 - 송신측이 한 번에 몇바이트씩 전송했는지 수신측이 알지 못함 - 네트워크로부터 혼잡제어를 받음 	<ul style="list-style-type: none"> - 비연결형 트랜스포트 서비스 - 연결 설정이나 연결종료 과정 없이 목적지로 바로 IP 데이터그램을 전송 - 데이터그램 단위로 송수신 : 송신측이 한 번에 전송한 데이터 크기만큼씩 수신측에서 읽음 - 데이터의 분실 확인, 전달 순서를 보장하지 않음 - 프로토콜 헤더가 TCP보다 작고 연결 지연이 없음 - 네트워크로부터 혼잡제어를 받지 않음

응용 계층

- 트랜스포트 계층에서 TCP 또는 UDP중 어느 것을 사용하는 지에 따라 분류

계층	사용 프로토콜	특징
응용 계층	http, CORBA, rsh, rcp, RPC	<ul style="list-style-type: none"> - 이미 작성된 네트워크 패키지를 활용 - 복잡한 서비스를 간단히 제공할 수 있음 - 전송 효율은 떨어질 수 있으나 프로그램 작성, 변경, 운영이 쉬움
트랜스포트 계층	TCP/IP, 유닉스 socket, Winsock	<ul style="list-style-type: none"> - 데이터그램 단위의 데이터 송수신을 처리함 - 종점간 연결관리, 흐름제어, 에러제어 등을 이용할 수 있음 - 인터넷 응용 프로그래밍의 기초가 됨
디바이스 드라이버 계층	NDIS, Raw 소켓 패킷 캡처	<ul style="list-style-type: none"> - LAN에서 프레임 단위의 송수신을 처리함 - 흐름제어, 에러제어 등은 사용자가 작성해야함

표 1.1 TCP와 UDP가 지원하는 응용 계층 서비스

인터넷과 서브네트워크

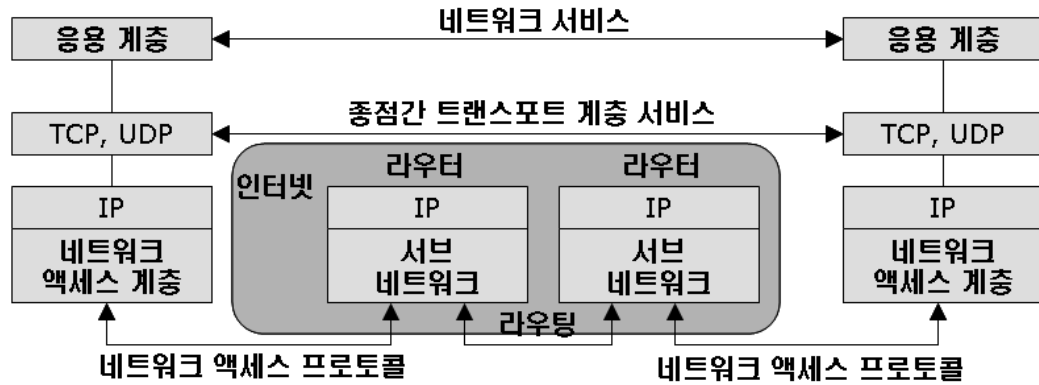


그림 1.4 TCP/IP와 서브네트워크 프로토콜

두 호스트 사이의 통신

- 호스트에 TCP/IP 프로토콜이 설치되어 있어야 함
- 호스트는 라우터들을 경유하여 서로 연결되어 있어야 함
- 호스트가 인터넷에 물리적 접속을 위해서는 네트워크 액세스 프로토콜이 필요함
 - ⇒ IP계층이 서브네트워크를 이용하기 위한 프로토콜이라고 할 수 있음
- 서브네트워크
 - 이더넷, 패킷 교환망, DSL, ATM(Asynchronous Transfer Mode) 등 실제 데이터를 전달해주는 네트워크

1.1.4 통신 프로토콜 표현 방식

필드 정의

통신 프로토콜의 내용을 정하기 위해 프로토콜 데이터 단위(PDU)의 헤더나 트레일러의 필드 이용

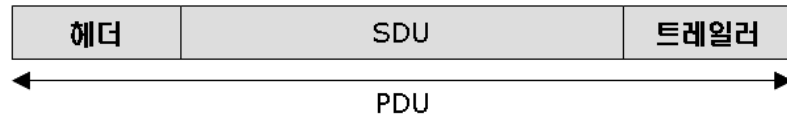


그림 1.5 특정 계층의 PDU와 SDU의 관계

파라미터 정의

통신 프로토콜을 표현하기 위해 미리 정해진 파라미터를 사용

상태 정의

통신 프로토콜에서 통신상태란 과거의 동작 결과를 바탕으로 얻어진 현재의 동작환경

동작 내용 정의

상태 정의와 밀접한 관계, ‘어떤 상태에서 어떤 입력을 받아 어떤 작업을 수행’ 식의 프로토콜 구현

필드 정의

- 서비스 데이터 단위(SDU)
 - 통신 프로토콜 계층에서 상위 계층으로 전달하는 데이터 부분
 - 사용자 데이터 또는 유료부하(payload)라고 함
 - PDU : SDU에 헤더와 트레일러가 추가된 것
- 헤더나 트레일러에는 그 계층의 프로토콜 처리를 위한 정보가 포함됨
- 헤더나 트레일러 크기를 여유 있게 정의하면 다양한 프로토콜을 쉽게 표현
- PDU중에 헤더+트레일러의 비율(오버헤드)이 증가하면 데이터 전송 효율이 떨어짐

파라미터 정의

- 파라미터의 값이 너무 크거나 작으면 통신 처리 성능이 떨어짐
- 경우에 따라 파라미터 값이 네트워크 상태에 따라 자동으로 바뀌게 설정

상태 정의

- 통신 서비스는 현재 자신의 상태에 따라 처리할 내용이 정해짐
 - 통신시작 전의 준비상태, 연결요청 상태, 연결 후 데이터 송수신상태, 에러상태, 정상종료 등

동작 내용 정의

- 입력의 종류
 - 상위계층으로부터의 명령 : 이 계층이 처리하도록 주어지는 작업
 - 하위 계층으로부터의 서비스 : 하위 계층이 이 계층으로 보내는 SDU 내용
 - 자체 이벤트 : 타이머의 종료, 카운터 횟수 만료 등의 이벤트

1.2 네트워크 프로그래밍의 분류

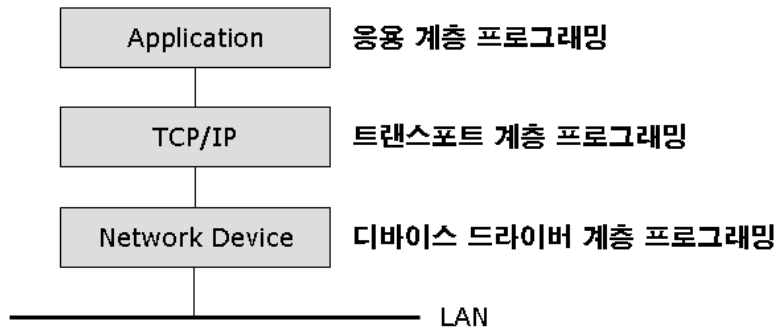


그림 1.6 네트워크 프로그래밍의 계층적 분류

디바이스 드라이버 계층 프로그래밍

- OSI의 계층2 이하의 인터페이스(링크 계층과 하드웨어 디바이스)를 통해 프레임 단위의 데이터 송수신을 직접 다룸
- 대표적인 API
 - LAN에서 MAC 프레임 단위의 송수신을 다루는 API
예) FTP사의 패킷 드라이버, 마이크로소프트사의 NDIS, 노벨사의 ODI
 - MAC 프로토콜의 종류와 LAN 카드 제조사에 무관하게 드라이버 계층의 네트워크 프로그램 작성할 수 있음
- 구체적인 송수신을 제어하거나 네트워크 상태 모니터링에 사용
- 흐름제어, 오류제어, 인터넷 주소 관리 같은 기능은 사용자가 별도 구현해야 함

트랜스포트 계층 프로그래밍

- TCP나 UDP와 같은 트랜스포트 계층의 기능을 직접 이용
- 호스트 사이의 연결 관리와 패킷 단위의 데이터 송수신을 직접 제어
- 대표적인 API
 - 소켓(socket) API
 - 예) UNIX의 BSD소켓, 윈도우 소켓
- 소켓 인터페이스는 BSD 유닉스에서 처음 보급되었으나 현재는 컴퓨터 기종, OS에 무관하게 지원함
- TCP/IP를 제공하는 컴퓨터에서 기본적으로 모두 지원함

응용 계층 프로그래밍

- 네트워크 유틸리티나 미리 만들어진 응용 계층 서비스 프로그램을 활용하는 방식으로 OSI 5-7계층을 이용하는 프로그래밍
- 패킷의 송수신을 구체적으로 제어하는 방식이 아닌 응용작업 단위의 동작을 네트워크를 통해 실행
 - 유닉스의 rsh(remote shell)이나 rcp(remote copy)를 이용하는 프로그래밍
 - 원격 컴퓨터에서 어떤 프로세스를 실행시키는 RPC(Remote Procedure Call) 프로그래밍
 - HTTP를 이용하는 HTML프로그래밍, 웹 프로그래밍, 미들웨어를 이용하는 분산 객체 프로그래밍
- 하위 계층의 동작(종점 호스트간의 연결 설정, 패킷 송수신, 흐름제어 등)을 구체적으로 제어 못함
- 복잡한 기능의 네트워크 서비스 프로그램을 짧은 시간 내에 작성할 수 있음

1.3 클라이언트-서버 모델

1.3.1 클라이언트-서버 모델의 정의

서비스를 제공하는 장비를 서버, 서비스를 이용하는 장비를 클라이언트라 함

2-tier 클라이언트-서버 모델

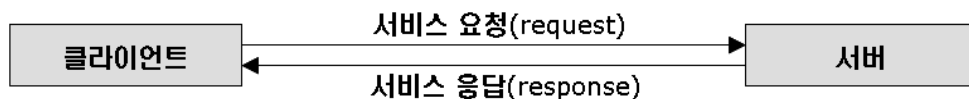


그림 1.7 2-tier 클라이언트-서버 모델

클라이언트-서버 모델

- 클라이언트는 서비스를 이용하기 위한 요청(request)을 서버로 보냄
- 서버는 서비스를 제공하기 위한 응답(response)을 클라이언트에게 보냄
- 일반적으로 서버를 먼저 설계하고 클라이언트는 서버가 제공하는 서비스를 이용하도록 설계함
- 서버의 기능
 - 요청을 보내온 클라이언트에 대한 인증(authentication)
 - 서버의 정보를 보호(security)
 - 여러 클라이언트에게 동시에 서비스 제공(concurrency)
 - 네트워크 오동작이나 잘못된 클라이언트의 요청에 안정적 실행(stability)

2-tier 클라이언트-서버 모델

- 클라이언트가 서버로 서비스 요청을 보내고 서버는 이에 대해 서비스 응답을 보내는 방식
- 대부분의 통신 프로그램(http, ftp, telnet, mail 등)이 이 방식으로 동작
- 병목 현상이 생기기 쉬운 단점
 - 클라이언트 수가 늘어날수록 서버에 트래픽 집중 현상 또는 처리 용량 부족 현상이 발생

■ fat 클라이언트

- 단점보안을 위해 네트워크 서비스를 제공하기 위한 기능의 일부를 클라이언트 측에 구현
- 서버의 처리 부담을 줄이는 장점이 있지만 프로그램의 업그레이드나 버전관리 등이 불편함

3-tier 클라이언트-서버 모델

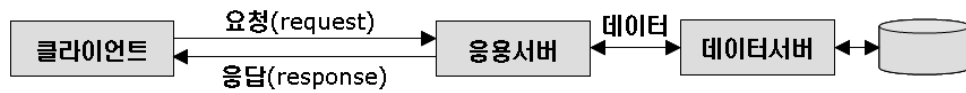


그림 1.8 3-tier 클라이언트-서버 모델

3-tier 클라이언트-서버 모델

- 2-tier 클라이언트-서버 모델의 문제점을 개선한 구조
- 서버의 기능을 응용서버와 데이터 서버로 나눔
 - tier1 : 클라이언트
 - tier2 : 응용서버
 - tier3 : 데이터서버
- 클라이언트는 응용서버에게 서비스 요청을 보내고 응용서버는 데이터서버로부터 필요한 데이터를 얻어 서비스를 처리한 후 결과를 클라이언트로 전송
- 클라이언트는 응용서버만 상대하며 데이터서버에 대해 알 필요 없음
- 같은 데이터를 여러 클라이언트가 동시에 요청할 때 응용서버에서 일괄처리함으로써 통신 부담 저하
- 클라이언트가 데이터서버를 직접 액세스하는 것이 효율적일수도 있음

n-tier 클라이언트-서버 모델

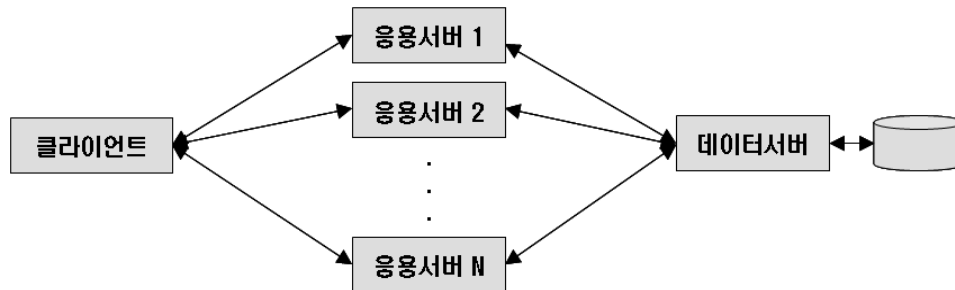


그림 1.9 n-tier 클라이언트-서버 모델

n-tier 클라이언트-서버 모델

- 3-tier 클라이언트-서버 모델을 확장한 것으로 여러 버전의 응용서버가 존재할 수 있는 모델
- 기본 동작은 3-tier 클라이언트-서버 모델과 같지만 응용서버가 여러 형태로 구현되는 모델
- 클라이언트가 필요에 따라 각각 다른 응용서버를 선택
- 서비스가 제공되는 도중에 새로운 응용서버를 추가
- 서비스의 다양성, 확장성, 업그레이드 등을 편리하게 제공
 - 단, 각 tier 사이의 인터페이스가 미리 명확하게 정의되어야함

P2P 모델

서버 또는 클라이언트의 역할을 담당할 컴퓨터를 미리 구분하지 않고 참가자에 따라 변화

순수 P2P모델, 하이브리드형 P2P모델이 있음

P2P 모델

■ 순수 P2P 모델

- 참가자들이 동등한 자격으로 서로 협력하여 정보를 이용하는 모델
- 자료 검색을 위해 주변 참가자에게 문의함, 이 과정이 자료를 찾을 때 까지 계속됨
- 동작이 단순하고 비효율적임

■ 하이브리드형 P2P 모델

- 순수 P2P 모델을 개선한 방식
- 인덱스 서버를 두어 자료를 찾을 때 인덱스 서버에 자료를 요청하고 인덱스 서버는 자료를 가진 참가자의 주소를 알려줌

1.3.2 서버 구현 기술

연결형과 비연결형 서버

연결형 서비스 : 종점간 연결 설정/해제, 데이터 송수신 등 세 단계의 절차를 거침

비연결형 서비스 : 종점간 연결 설정/해제 작업 없이 바로 데이터를 주고받는 방식

Stateful과 Stateless 서버

Stateful 서버 : 클라이언트와의 통신 상태를 계속 추적하여 서비스를 제공

Stateless 서버 : 상태 정보를 이용하지 않고 독립적인 요청에 의해 서비스를 제공

Iterative와 Concurrent 서버

Iterative 서버 : 클라이언트의 요청을 순서대로 처리

Concurrent 서버 : 클라이언트 요청을 동시에 처리

연결형과 비연결형 서버의 특징

■ 연결형 서버

- TCP와 같은 연결형 프로토콜 사용
- 데이터의 안정적인 전달 보장
- 모든 클라이언트와 연결 개설
- 클라이언트 수가 많아질수록 서버의 부담 가중
- 회선교환 서비스, TCP 프로토콜, telnet, ftp 등이 있음

■ 비연결형 서버

- UDP와 같은 비연결형 프로토콜을 사용
- 데이터의 전달을 보장할 수 없음
- 클라이언트마다 연결을 설정하지 않음 (서버 부담이 적어짐)
- 이더넷 프로토콜, IP 계층 프로토콜, UDP 프로토콜, 웹 서비스 등이 있음

Stateful과 Stateless 서버의 특징

- Stateful 서버
 - 서버가 클라이언트와의 통신 상태(state)를 계속 추적하며 이 정보를 서비스 제공에 이용
 - 메시지의 양을 줄일 수 있음
 - 오류에 의한 오동작 가능성이 있음
- Stateless 서버
 - 독립적인 request에 의해 서비스 제공
 - 클라이언트로부터 새로 도착한 request에만 의존하여 서비스를 제공
 - 메시지의 길이가 stateful 서버의 경우보다 길어질 수 있음
 - 서버의 안정적인 동작

Iterative와 Concurrent 서버의 특징

- Iterative 서버
 - 클라이언트의 서비스 요구를 순서대로 처리
 - 각 서비스 처리시간이 충분히 짧을 때 사용
 - 서버 프로그램 구현이 비교적 간단함
- Concurrent 서버
 - 여러 클라이언트의 요구를 동시에 처리
 - 각 서비스 처리시간이 불규칙적이거나 길 때 사용
 - 서버 프로그램 구현이 복잡함
 - 다중처리 기능이 필요

2장

TCP/IP 프로토콜

2.1 네트워크 액세스 계층

2.2 IP 프로토콜

2.3 TCP 프로토콜

2.4 UDP 프로토콜

Overview

- 네트워크 액세스 계층에서 사용되는 서브네트워크에 대한 지식 습득
- TCP, IP, UDP 프로토콜에 대한 기본적인 지식 습득

2.1 네트워크 액세스 계층

2.1.1 이더넷

이더넷 프레임 구조

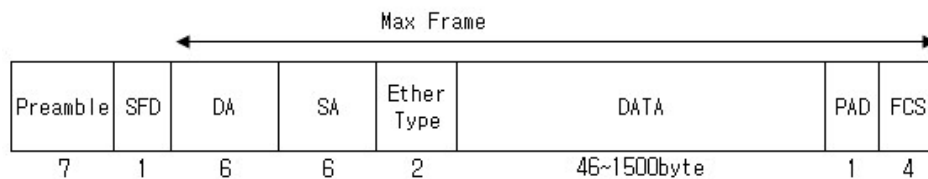


그림 2.1 이더넷 프레임 구조

이더넷 프레임 구조

- Preamble(8비트) - 수신측에서 하드웨어 비트 동기를 맞추기 위한 준비 신호
- SFD(Starting Frame Delimiter) - 다음 바이트열이 프레임의 시작을 알림
- DA(Destination Address, 6비트)
 - 목적지 MAC주소
 - 앞의 3바이트는 카드회사 식별코드이며 나머지 3바이트는 개별적인 목적지 식별용 NIC카드임
- SA(Source Address, 6비트)
 - 송신측 MAC주소
 - 자신의 주소를 ROM에 기록하여 초기화시 ROM에서 읽어 레지스터에 저장함
 - 프레임 송신시 이 레지스터를 읽어 프레임의 SA에 삽입함
- EtherType - MAC 프레임 다음의 데이터 부분에 상위 프로토콜의 종류를 표시함
- Data
 - 상위 프로토콜이 위치
 - 최대 허용길이(MTU)는 1500바이트임
 - DA부터 FCS까지 전체길이가 64바이트 이상이어야 한다는 규정을 준수해야 함
- PAD - 최소길이 규정을 만족하지 못할 경우 '0'으로 채움
- FCS
 - 프리앰블과 SFD를 제외한 MAC프레임의 비트열의 에러를 검사함

MAC 주소 구조

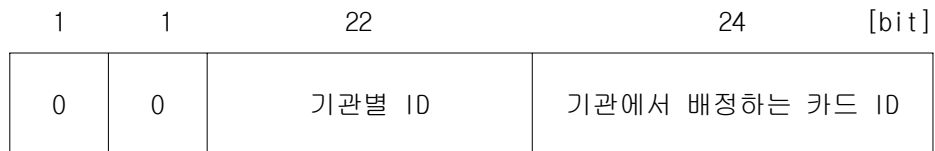


그림 2.2 MAC 주소 구조(48비트)

MAC 주소 구조

- 처음 두 비트 - 항상 0
- 다음 22비트 - LAN 카드 제조 회사별로 할당된 주소
- 뒤의 24비트 - LAN 카드마다 유일하게 제조사가 부여하는 번호

이더넷의 특징

사용 프로토콜
케이블
스위칭 이더넷

사용 프로토콜

- CSMA/CD(Carrier Sense Multiple Access/Collision Detection)
 - ⇒ 접근 제어(MAC) 프로토콜로 사용함
 - 프레임을 전송하려는 스테이션은 먼저 채널감지(CS : Carrier Sense)를 통해 다른 장비가 전송중인지 확인함
 - 채널이 사용 중이지 아닌 때에만 전송을 시도
 - 프레임 전송 중에 다른 장비가 전송을 하여 충돌이 나는지 감시(Collision Detection)
 - 충돌이 나면 즉시 전송을 중지

케이블

- category 5 UTP(Unshielded Twisted Pair) 케이블을 널리 사용
 - 4pair(8가닥)로 되어 있으며 10Mbps 이더넷(10BaseT)에서는 두 pair만 사용
 - 동작속도 100Mbps인 고속 이더넷(100Base4T)에서는 네 pair를 모두 사용
 - 한 pair는 충돌감지용, 나머지 세 pair는 각각 33.3Mbps 속도로 데이터 전송

스위칭 이더넷

- 일반 허브를 스위칭 허브로 교체하여 허브에 연결된 장비가 각각 10Mbps 대역을 모두 사용
- 스위칭 이더넷의 속도를 증설하여 채널 속도 100Mbps인 것이 널리 사용
- 현재는 이를 개선한 1Gbps 또는 10Gbps의 확장된 기가비트 이더넷 사용

2.1.2 PPP

PPP 프레임 구조

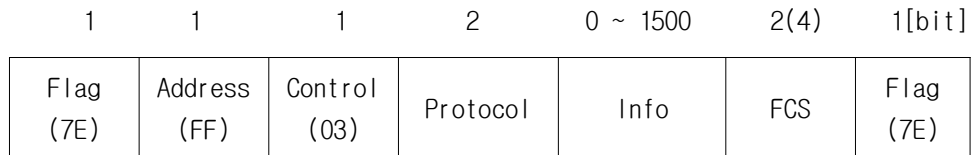


그림 2.3 PPP 프레임 구조

PPP 프레임 구조

- PPP(Point-to-Point Protocol)는 전화회선, xDSL(Digital Subscriber Line) 같은 일 대 일로 연결된 회선을 통해 IP 데이터그램을 송수신하는 통신 프로토콜
- 프레임구조
 - 프레임의 시작과 끝은 플래그로 0x7E를 사용함
 - Address와 Control은 각각 0xFF와 0x03으로 고정됨
 - Protocol 필드는 상위 계층에 전달하는 정보(Info)의 종류 구분에 사용함
 - FCS(Frame Check Sequence)는 프레임 전송 중 발생하는 비트 에러를 검출함

2.1.3 기타 서브네트워크

X.25 패킷 교환망

OSI 표준 1-3계층을 따르는 패킷 교환 프로토콜

ISDN

전화망과 패킷 교환 방식의 데이터 통신망을 하나의 네트워크에서 서비스하는 네트워크

ATM망

고속 패킷 교환 기술

무선전화망(IS-95)

X.25 패킷 교환망

- 주로 공중 데이터 통신망에서 사용됨

ISDN

- 내부동작은 X.25와 유사한 패킷 교환 방식을 사용
- 외형적으로는 디지털 회선 교환 서비스를 통합하여 제공

ATM망

- X.25와 달리 OSI 계층 3이 아닌 물리 계층에서 직접 패킷 스위칭
- 일반 가입자 전송용이 아닌 교환기 사이의 고속 채널 교환 방식으로 사용됨
- 모든 데이터를 53바이트의 고정된 길이의 셀(cell) 단위로 전송 및 교환
 - 교환기를 간단하고 빠르게 만들 수 있음
 - 여러 종류의 트래픽을 수용하기에 편리함

무선전화망

- 무선 전화망을 통해 인터넷에 접속하는 것을 무선 인터넷이라 함
- 유선 채널에 비해 대역폭이 좁고 채널 비용이 비쌈
 - 통신량을 줄이는 것이 중요
- 핸드폰, 소형 정보기, 노트북에서 사용됨

2.2 IP 프로토콜

IP 프로토콜의 특징

IP 프로토콜의 특징

- OSI 7계층의 관점에서 보면 IP계층은 주소와 라우팅을 담당하는 OSI 계층 3에 해당함
- IP 계층은 서브네트워크를 이용해 IP 데이터그램을 임의의 호스트 사이에 전달하는 기능
- 수많은 임의의 호스트를 4바이트의 IP 주소만으로 찾고 데이터그램을 전달가능하게 함
- 비연결 방식으로 데이터그램을 전달
- 데이터그램의 분실, 중복, 전달 순서 바뀜, 비트 에러 등이 발생해도 IP 계층에서는 데이터그램 에러 확인이나 재전송 등을 하지 않음
- 데이터그램을 단순히 목적지로 전달만 하고 전송 결과 확인은 하지 않음

2.2.1 IP 데이터그램 구조

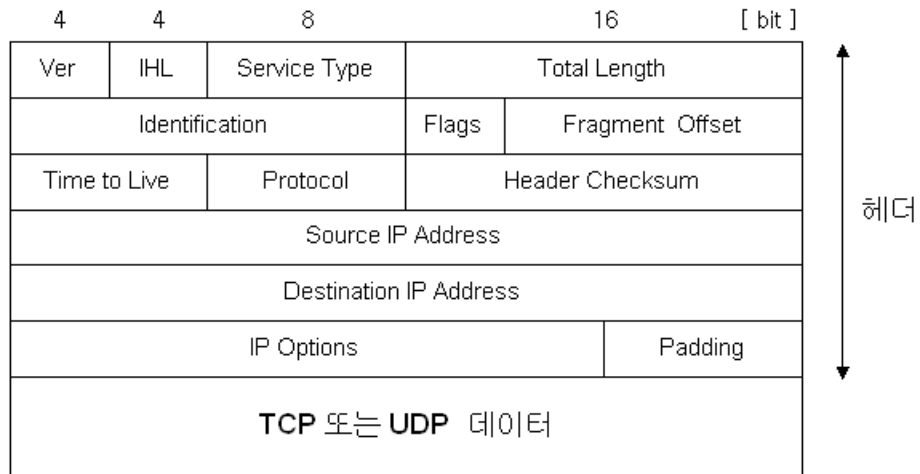


그림 2.4 IP 데이터그램 구조

데이터그램 구조

- 한 줄(row)
 - 4바이트(32비트)의 크기

- 한 열(column)
 - IP헤더는 보통 20바이트
 - IP가 전달하는 데이터(TCP, UDP, ICMP 메시지)는 IP 헤더 뒤에 연속하여 전송됨

IP 데이터그램 헤더

필드명		길이(비트)
Ver (Version)		4
IHL		4
Service Type		8
Total Length		16
Identification		16
Flags	미사용	1
	DF	1
	More	1
Fragment Offset		13
TTL (Time To Live)		8
Protocol		8
Header Checksum		16
Source IP Address		32
Destination IP Address		32
IP Options		가변
Padding		가변

표 2.1 IP 데이터그램 헤더

IP 데이터그램 헤더

- Ver
 - IP 버전 값을 표시하며 현재는 4의 값을 가짐

- IHL (Internet Header Length)
 - 헤더 길이를 4바이트 단위로 표시
 - 최소 값은 5 (헤더의 최소 크기는 20 바이트)

- Service Type
 - 서비스 클래스 지정
 - 보통 0으로 지정

- Total Length
 - 헤더를 포함한 IP 데이터그램의 전체 크기를 바이트 단위로 나타냄
 - 16비트로 표현하므로 최대 값은 65535

■ Identification

- 상위 계층이 전송한 큰 메시지가 여러 데이터그램으로 단편화 되어 전송되었을 때 수신측에서 원래 메시지를 재구성할 때 사용하는 번호
- 한 메시지를 구성하는 데이터그램들은 모두 같은 Identification 값을 가짐

■ Flag

- 첫 번째 비트 : 사용하지 않음, 항상 0
- 두 번째 비트 : 단편화 금지 플래그 DF(don't fragment) 비트
 - DF = 0 : 데이터그램의 단편화를 허용, 중간의 라우터가 필요에 따라 수행
 - DF = 1 : 데이터그램의 단편화를 허용치 않음
- 세 번째 비트 : More 비트
 - 라우터가 More = 1인 데이터그램들을 재조립하여 더 큰 크기의 IP 데이터그램으로 전송
 - More = 1 : 이 데이터그램과 다음에 전송되는 데이터그램이 단편화되어 전송됨을 나타냄
 - More = 0 : 한 메시지의 마지막을 구성하는 데이터그램 또는 메시지가 한 데이터그램임

■ TTL

- 데이터그램들이 인터넷 내에서 계속 돌아다니는 것을 방지하기 위하여 사용
- 보통 hop counter(노드를 지나는 횟수) 값을 사용
- 한 노드를 지날때마다 1씩 감소하며 0이 되는 노드에서 이 데이터그램을 삭제
- TTL값이 너무 클 때
 - 목적지 주소에서 에러발생시 데이터그램이 네트워크상에 오래 남아 트래픽 증가
- TTL값이 너무 작을 때
 - 데이터그램이 먼 거리를 돌아 갈 때 목적지에 도착 못함
- 디폴트로 64의 값을 가지며 멀티캐스트 같이 트래픽이 많이 발생할 때는 TTL을 작게 함

■ Protocol

- 데이터그램에 실려 있는 데이터를 처리할 상위 계층 프로토콜을 지정
 - TCP : 6
 - UDP : 17
 - ICMP : 1

■ Header Checksum

- IP 헤더의 비트 에러 검출을 위해 사용
- 헤더를 16비트 단위 크기로 나누고 각 16비트를 숫자로 취급하여 차례로 더해 Header Checksum에 실어 보냄
- one's complement로 계산하여 수신측 결과와 다를 경우 에러 발생이므로 데이터그램 삭제

■ Source IP Address

- 송신지 IP 주소

■ Destination IP Address

- 수신지 IP 주소

■ IP Options

- 옵션 선택
- 보통 사용되지 않음

■ Padding

- 32비트 단위로 헤더의 길이를 맞춤
- 보통 사용되지 않음

IP 주소

IP 주소의 구성

IP 주소 체계의 단점

IP 주소가 netid와 hostid 두 부분으로 구성된 점

IP 주소의 구성

- IP 주소는 4바이트 크기를 가짐
- dotted decimal IP 주소
 - IP 주소를 보기 쉽게 표시
 - IP 주소를 구성하는 4바이트를 10진수로 표시
 - 예) 11010010 01110011 00110001 11110100 -> 210.115.49.244
- netid 필드
 - 네트워크를 구분
- hostid 필드
 - 한 네트워크 내에서 호스트를 구분
- 각 필드에서 사용하는 비트 수의 크기에 따라 네 가지 클래스로 분류

IP 주소 체계의 단점

- 호스트가 다른 네트워크로 이동하면 netid부분이 새로 접속된 네트워크의 netid로 변경
 - 컴퓨터의 IP 주소가 달라짐

IP 주소의 종류

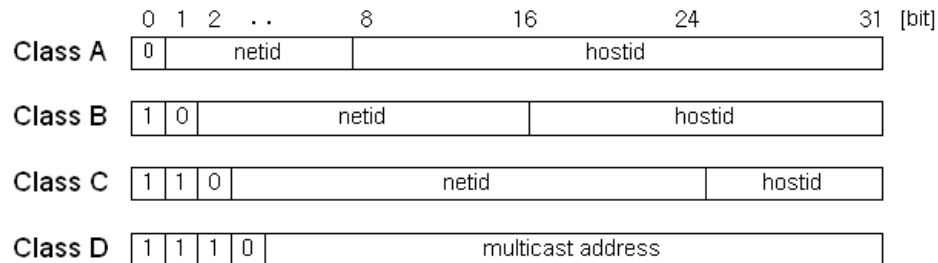


그림 2.5 IP 주소의 종류

클래스 C 주소의 특징

방송용과 자기 자신을 가리키는 주소가 있음

IP 주소의 종류

- 클래스 A 주소
 - IP 주소 첫 번째 바이트의 첫 비트가 0이고 나머지 7비트가 netid
 - 뒤의 세 바이트가 hostid
 - 클래스 A 주소의 netid는 약 2^{24} 대의 호스트를 수용
- 클래스 B 주소
 - IP 주소 첫 번째 바이트의 처음 두 비트가 10이고 나머지 6비트와 두 번째 바이트가 netid
 - 뒤의 두 바이트가 hostid
 - 클래스 B 주소의 netid는 약 2^{16} 대의 호스트를 수용
- 클래스 C 주소
 - IP 주소 첫 번째 바이트의 처음 세 비트가 110이고 나머지 5비트와 2, 3번째 바이트가 netid
 - 마지막 한 바이트가 hostid
 - 클래스 C 주소의 netid는 약 2^8 대의 호스트를 수용
- 클래스 D 주소
 - IP 주소 첫 번째 바이트의 처음 네 비트가 1110
 - 멀티 캐스트 주소로 사용

클래스 C 주소의 특징

- 8비트의 hostid로 구분하는 총 $2^8 = 256$ 개의 호스트를 가짐
 - 모두 1 : 방송용
 - 모두 0 : 자기 자신
- 위 두 개를 제외한 254개만 사용 가능

2.2.2 IP 계층 기능

MTU(Maximum Transmission Unit)

서브네트워크가 한 번에 전달할 수 있는 데이터그램의 최대 크기

path MTU

종점 호스트 사이에 존재하는 서브네트워크들의 MTU중 최소 MTU값

MTU

- 이더넷의 경우 MTU는 1500바이트
- PPP의 MTU는 전송매체에 따라 가변적이지만 전화선일 경우 주로 576바이트를 사용
- IP 계층이 상위 계층으로부터 받은 메시지의 크기가 path MTU보다 큰 경우
 - 이를 여러개의 IP 데이터그램으로 단편화하여 전송
 - 수신측에서는 단편화된 데이터그램을 재조립
- 데이터그램의 identification 필드는 단편화 된 다수의 패킷을 원래대로 재구성하기 위해 존재
- 전체 메시지를 구성하는 데이터그램 중 하나라도 손실 될 경우
 - 최종 수신측에서 전체 메시지를 버림
 - 메시지의 재조립은 라우터에서는 실행되지 않음

path MTU

- IP 헤더의 플래그 DF가 1이면서 IP 데이터그램의 크기가 path MTU보다 클 경우
 - 중간의 라우터에서 에러가 발생하며 송신측에게 이를 알려줌
- 일반 데이터그램의 DF 비트는 항상 0으로하여 필요시 라우터에서 단편화함
- DF 비트는 path MTU의 크기를 알아내는 path MTU discovery 프로토콜에서 사용됨
- TCP 계층에서 일정 크기의 데이터그램을 DF = 1로 하여 전송시
 - 목적지까지 전달되지 못하고 에러 발생하면 데이터그램의 길이를 조금씩 줄여서 재전송
 - 전송이 성공할 때까지 반복하여 path MTU를 찾음

라우팅

IP 데이터그램이 목적지 호스트까지 전달되기 위해 거쳐야 할 라우터를 정해주는 기능

ARP(Address Resolution Protocol)

LAN 내에서 특정 IP 주소를 가지고 있는 호스트의 MAC 주소를 알아내는 프로토콜

라우팅

- 현재 인터넷이 널리 사용 될 수 있는 이유 중 하나로 안정적인 라우팅 동작이 있음
- 라우터
 - 라우팅을 처리하는 장비
- 라우팅 알고리즘
 - 최적의 라우팅 경로를 알아내는 알고리즘
- 라우팅 테이블
 - 라우팅 알고리즘으로 얻은 라우팅 정보를 정리한 것

ARP

- LAN에 접속된 호스트에게 IP 데이터그램을 전달하려면 그 장비의 MAC 주소를 알아야함
 - LAN에서는 모든 데이터가 MAC 프레임에 실려서 전달
 - IP 주소로부터 해당 호스트의 MAC 주소를 알아내는 절차를 ARP라고 함
 - IP 주소가 아닌 48비트의 MAC 주소가 사용되기 때문에 ARP 프로토콜이 필요

ARP 동작 순서

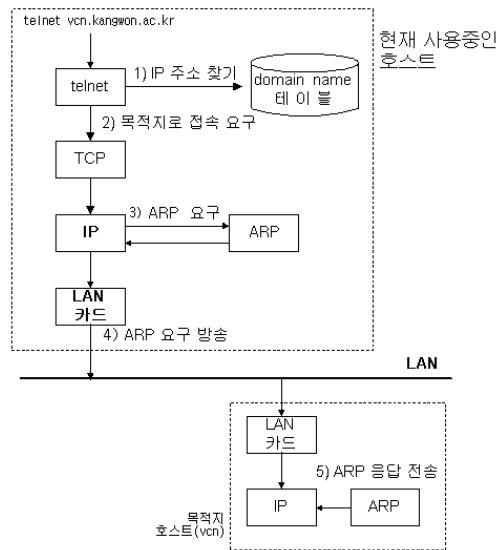


그림 2.6 ARP의 동작

RARP(Reverse ARP)

ARP의 역과정

ARP의 동작 순서

- telnet vcn.kangwon.ac.kr 이라는 명령을 내렸을 시
 - 응용 프로그램 telnet은 먼저 vcn.kangwon.ac.kr 호스트가 210.115.49.242 임을 찾음
→ 네임 서비스
 - telnet은 이 IP 주소를 TCP에게 알려주고 이곳으로 접속 하도록 요청
 - IP 계층은 목적지 호스트의 netid와 자신의 netid가 같은지 비교
→ 같은 LAN에 있으면 LAN을 통해 바로 telnet 서비스 요청을 전송
 - 목적지의 MAC 주소를 알아내기 위해 ARP를 구동
 - ARP에서 목적지 호스트의 MAC 주소를 찾기 위해 ARP request 패킷을 LAN 내의 모든 장비에 방송
 - 목적지 호스트(vcn.kangwon.ac.kr)만 ARP request에 자신의 MAC주소를 응답
 - 만약 같은 LAN에 있지 않으면 telnet 서비스 요청을 디폴트 게이트웨이를 통해 외부로 요청
- 동일한 목적지 호스트에 IP 데이터그램을 연속으로 보낼 때
 - 계속 ARP를 사용하면 트래픽이 증가됨
 - ARP로 얻은 최근 정보를 캐시에 기록되어 있어 이를 잠시 재사용하여 트래픽 문제 해결

RARP

- 48비트 MAC 주소로부터 그 장비의 IP 주소를 알아내는 프로토콜

2.3 TCP 프로토콜

2.3.1 TCP 특징

연결형 서비스 제공

종점 호스트 사이에 일 대 일 연결을 제공

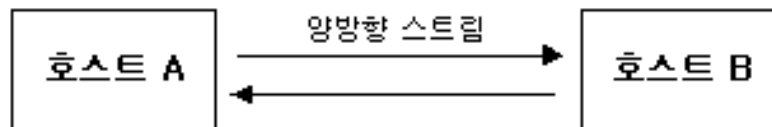


그림 2.7 TCP 연결의 양방향 스트림 생성

연결형 서비스 제공

- 연결설정, 데이터 송수신, 연결종료 과정이 필요함
- 두 호스트 사이에 TCP 연결 설정이 이루어지면 두 개의 스트림이 생성되어 양방향 통신함
- 호스트 A가 호스트 B로 연결요청을 한 경우에도 호스트 B가 호스트 A로 데이터 전송 가능

신뢰성 있는 데이터 전달

신뢰성 있는 종점간 데이터 송수신을 보장함

확인응답(Ack), 체크섬(Checksum), 재전송, 흐름제어 등을 사용

신뢰성 있는 데이터 전달

■ 확인응답(Ack)

- 상대방 TCP에게 잘 전달되었는지를 확인하기 위해 사용
- 데이터를 수신한 상대방은 데이터에 에러가 없고 순서에 문제가 없을시 송신측에 Ack를 보냄
- 에러 확인은 체크섬, 순서확인은 순서번호를 사용함
- piggyback기능
 - 데이터를 보낼 때 마다 Ack를 전송하면 불필요한 Ack가 발생됨
 - 수신측은 즉시 Ack를 보내지 않고 약간의 시간지연을 둠
 - 지연시간동안 보낼 데이터가 생기면 데이터에 Ack를 포함시켜 전송

■ 체크섬(Checksum)

- 데이터 전송 중에 발생한 에러를 검출
- 체크섬 에러 발견시 데이터를 버리고 Ack를 보내지 않음
- 따로 에러 상황을 송신측에 알려주지는 않음

■ 재전송

- 일정 시간동안 Ack가 오지 않을 경우 전송했던 데이터를 재전송함
- 재전송을 위해 송신측은 타이머를 사용함
 - 타이머 시간동안 Ack가 오지 않을 경우 재전송
- 전송 타이머 값은 RTT(Round Trip Time)에 비례하여 정해지며 횟수에도 제한을 둠
 - 일정 회수 이상 Ack가 없을시 종점간 연결을 종료함

■ 흐름제어

- 수신측의 사정에 의한 흐름제어

- 수신버퍼의 여유 크기를 고려하여 상대방이 전송할 수 있는 데이터량을 바이트 단위로 알려줌
- 상대방이 너무 많은 데이터를 보내지 못하게 하는 흐름제어
- 수신 할수 있는 데이터량을 Window필드(16비트)를 통해 송신측에 알려줌

- 송신측의 사정에 의한 흐름제어

- 송신측에서는 송신버퍼가 부족하면 `write()`, `send()`같은 쓰기 함수가 블록됨

스트림형 서비스 제공

송신측에서 전송된 데이터는 바이트 단위로 차례대로 수신측에 전달됨

스트림형 서비스 제공

■ 스트림 서비스

- 송신된 데이터를 수신측에서 차례대로 읽을 수 있게 제공되는 통신 채널 서비스
- 수신측에서는 송신된 데이터를 한 번에 읽거나 여러 번으로 나누어 읽을 수 있음
- 송신측에서 `write()`등을 호출하여 전송한 데이터들의 경계를 수신측에서는 알 수 없음
- 송신측이 전송한 데이터 경계를 수신측이 알려면 전송 데이터 크기를 미리 정해야함

2.3.2 TCP 헤더

세그먼트

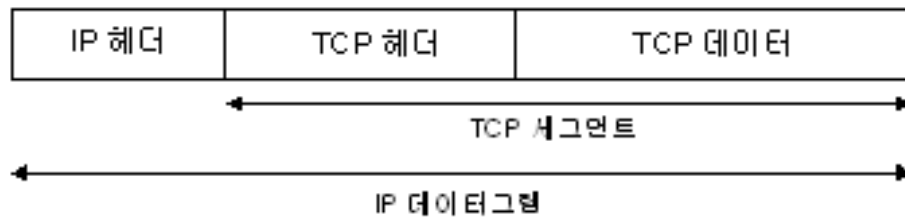


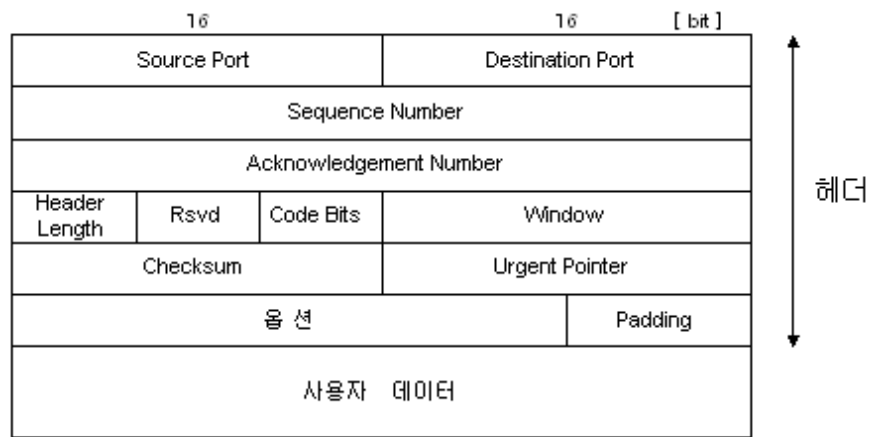
그림 2.8 IP 데이터그램과 TCP 세그먼트

세그먼트

- TCP 계층이 IP 계층으로 내려보내는 데이터 단위
 - 종점간 TCP들 사이에 송수신되는 데이터 단위

- 세그먼트의 구성
 - TCP 계층에서 프로토콜 처리를 위해 필요로 하는 헤더와 응용 프로그램으로부터 받은 데이터로 구성됨

TCP 세그먼트와 헤더 구조



Rsvd : Reserved

그림 2.9 TCP 세그먼트와 헤더 구조

TCP 세그먼트와 헤더 구조

- Source Port와 Destination
 - 각각 송신 및 수신측의 포트번호를 기록

TCP 헤더

필드명		길이(비트)
Source Port		16
Destination Port		16
Sequence Number		32
Ack Number		32
Header Length		4
Psvd		6
Code Bits	URG	1
	ACK	1
	PSH	1
	RST	1
	SYN	1
	FIN	1
Window		16
Checksum		16
Urgent Pointer		16

표 2.2 TCP 헤더 내용

TCP 헤더

- Source Port
 - 송신측의 응용 프로세스를 구분하는 포트번호
- Destination Port
 - 수신측의 응용 프로세스를 구분하는 포트번호
- Sequence Number 필드
 - 세그먼트에 실려 있는 데이터의 첫 번째 바이트의 순서번호 기록
 - 수신측에서 데이터가 정상적으로 도착하는지 확인하는데 사용
- Ack(Acknowledgement) Number
 - ACK 플래그가 1일경우에만 의미가 있고 일반적으로 ACK 비트는 1로 set 되어 있음
- Header Length
 - 헤더의 크기를 4바이트 단위로 나타냄
 - TCP에서 옵션을 사용하지 않는 경우 TCP 헤더의 크기가 20 바이트이므로 5로 set 됨

■ Rsvd

- 현재 사용되지 않는 값으로 0으로 set 됨

■ Code Bits

- URG : Urgent Pointer가 유효한 값을 나타냄
- ACK : Ack Number에 들어 있는 값이 의미 있는 값을 나타냄
- PSH : 이 데이터를 가능한 신속히 응용에 전달하도록 함
- RST : 연결을 reset할 때 사용됨
- SYN : TCP 연결을 시작할 때 사용됨
- FIN : TCP 연결을 종료 할 때 사용됨

■ Window

- 수신측이 현재 수신 가능한 데이터 버퍼 크기를 바이트 단위로 나타냄

■ Checksum

- pseudo 헤더
→ TCP 세그먼트 전체 + IP 헤더의 후반부 12바이트
- pseudo 헤더에 대한 에러 검출 코드임

■ Urgent Pointer

- URG 비트가 1일 때의 의미
- 현재 세그먼트에 포함된 긴급 데이터의 마지막 위치를 가리키는 오프셋이 저장
- 긴급 데이터는 Sequence Number 위치부터 시작하여 Urgent Pointer 바이트 만큼 범위에 속함

2.3.3 TCP 연결설정

3-way 핸드셰이크(handshake)

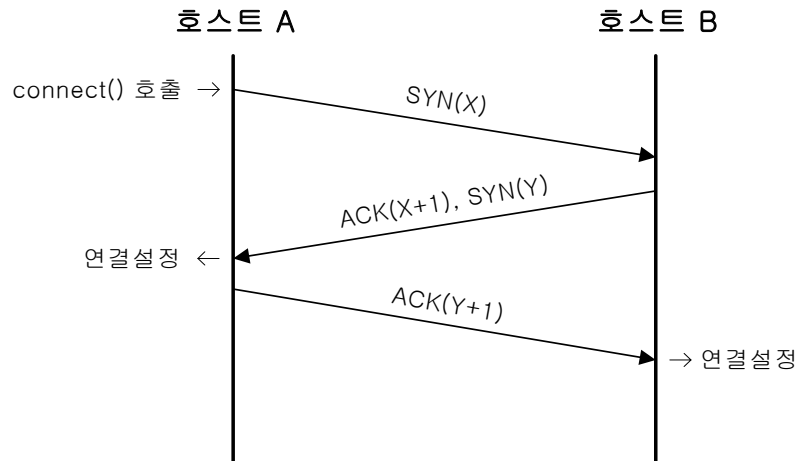


그림 2.10 TCP 연결설정 절차

3-way 핸드셰이크

- 3-way
 - 최초 연결요청에 상대방이 확인하면서 다시 연결요청하고 이에 대해 처음의 연결요청 확인
 - 3회의 통신이 이루어져야함
- 연결설정 절차
 - 연결요청 측에서 SYN 비트를 세트하고 순서번호를 랜덤한 정수 X로 한 연결요청을 보냄
 - 상대방은 Ack를 위해 ACK 비트를 세트하고 Ack Number에 (X+1)을 기록하며 SYN 비트도 세트한 후 다음 자신이 사용할 랜덤한 순서번호 Y를 지정하여 보냄
 - 최초의 연결요청자가 ACK(Y+1)로 응답하면 연결됨
 - 이때 SYN세그먼트의 크기가 1바이트이므로 X+1 또는 Y+1을 Ack 함
- X와 Y값을 랜덤하게 정하는 이유
 - 예측가능한 일정한 값을 사용하면 연결설정 동안 해킹 공격을 받을 수 있음
- TCP 연결요청에 ACK가 오지 않으면 잠시 기다린 후 재시도함

2.3.4 TCP 연결종료

4-way 핸드셰이크

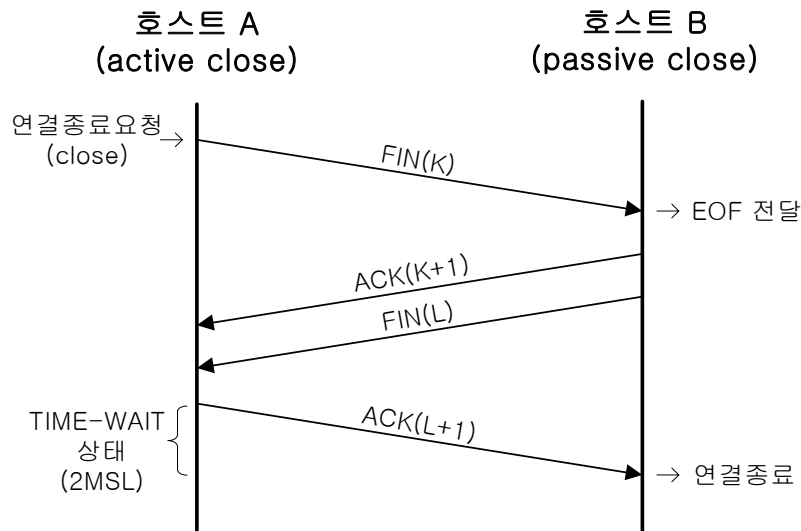


그림 2.11 TCP의 연결종료 절차

4-way 핸드셰이킹

- 연결 종료의 원인
 - TCP 응용 프로그램에서 해당 소켓에 대해 `close()`를 호출한 경우
 - 응용 프로그램이 종료된 경우
- 연결 종료 절차
 - TCP는 상대방으로 FIN 플래그를 세트하여 전송
→ 더 이상 전송할 데이터가 없다는 뜻
 - 이에 대한 ACK를 받음으로써 송신측 방향의 채널이 닫힘(half close)
 - FIN을 받은 측에서는 통상적으로 자신도 FIN을 보내어 양방향 채널을 모두 종료
- 처음 FIN을 보낸 측(그림 2.11의 호스트 A)
 - active close를 수행함
- FIN을 받은 측(그림 2.11의 호스트 B)
 - passive close를 수행함

TIME-WAIT 상태

Active close를 한 TCP가 FIN을 받고 이 FIN에 대한 ACK를 보내기 전에 잠시 머문 상태

TIME-WAIT 상태

- TIME-WAIT 상태에 있는 동안 지금까지 사용했던 포트번호를 재사용할 수 없도록 제한됨
- TIME-WAIT 상태를 두는 이유 1
 - 연결종료를 신청한 측(Active close)에서 ACK(L+1)까지 안전하게 전송해야 양방향 채널 종료
 - 만약 ACK(L+1) 메시지가 사라지면 상대방(Passive close) FIN(L)을 재전송하게 됨
 - 이때 TIME-WAIT 상태에 머물지 않았다면 FIN(L)에 대한 ACK(L+1)을 보낼 수 없음
 - TIME-WAIT 시간은 종료를 모두 완료하기 위해서 기다리는 시간
- TIME-WAIT 상태를 두는 이유 2
 - 연결종료 직후에 같은 IP 주소와 포트번호로 새로운 연결요청을 허용할 경우
 - 과거의 연결에 사용되었던 세그먼트가 뒤늦게 도착하는 것을 잘못 수신할 수 있음
 - 고의적인 해킹 공격 방지를 위해서 필요
- 2MSL
 - TIME-WAIT 상태의 다른 이름
 - TIME-WAIT 상태에서 기다리는 시간이 2*MSL(maximum segment lifetime)시간임
 - MSL은 네트워크 내에서 세그먼트가 남아있을 수 있는 최대 예상 시간
 - 대부분의 시스템은 30초에서 2분사이의 MSL값을 가짐
 - TIME-WAIT 상태에 있는 동안 과거에 전송된 세그먼트가 도착시
 - 이 세그먼트는 버려지고 2MSL 타이머는 재시작 됨

- TIME-WAIT 시간 이후에 세그먼트 도착시
 - 세그먼트는 무시되고 송신측으로는 리셋(RST) 비트를 세트하여 전송함
- Passive close 측에서는 TIME-WAIT 상태를 갖지 않음
 - FIN(L)을 보낸 후에 상대방으로부터 ACK(L+1)을 받으면 TCP 초기 상태로 감
- 클라이언트-서버 모델에서 서비스 사용이 종료될 때
 - 클라이언트는 active close를 신청하는 것이 바람직함
 - 서버측이 TIME-TIME 상태에 들어가지 않고 바로 같은 포트번호를 사용해서 다른 서비스 요청에 응답할 수 있음

리셋

정상적인 연결종료(FIN)를 사용 하지 않고 RST를 사용해 즉시 연결 종료함

리셋

- 정상적인 TCP 연결설정이 되지 않은 상태에서 데이터 수신시
 - TCP는 상대방으로 RST 비트를 세트하여 전송
 - 통신 중에 상대방 컴퓨터가 재부팅된 경우에도 발생
- 연결요청을 기다리고 있는 포트(서버 프로그램)가 없는 상태시
 - 연결요청을 받아도 상대방으로 리셋을 전송
- RST를 사용하여 즉시 종료(abortive release)를 할 경우
 - 아직 전송되지 않은 데이터에 대한 안전한 송신이 보장되지 않음
- RST를 수신한 측에서는 ACK를 보내지 않아도 됨
 - 수신측 TCP에서는 정상종료가 아닌 에러에 의한 연결 종료라고 응용 프로그램에 보고함
- TCP 연결종료 시에 FIN대신 RST를 전송 하려면 linger 소켓 옵션을 사용
 - SO_LINGER 옵션을 지정하면 연결 종료시 즉시 또는 옵션에서 정한 일정한 시간 후에 RST전송

2.3.5 데이터 송수신

MSS(maximum segment size)

IP 계층에서 단편화를 피하기 위해 지정한 최대 세그먼트 크기

MSS

- IP 계층 및 라우터에서 데이터그램을 발생하지 않도록 TCP에서는 종점간의 path MTU를 미리 알아내고 세그먼트의 크기를 “path MTU - IP와 TCP의 헤더크기” 정하는 것이 최적의 선택임
- MSS 값은 TCP 연결설정 시에 상대방과 협약함
 - SYN 전송시 자신이 원하는 MSS 값을 상대방에게 알려줌
 - 상대방 호스트는 자신의 MSS 값과 받은 MSS 값을 비교해서 두 MSS 값중 작은 값을 리턴함
 - 디폴트 MSS 값은 536바이트
 - 이경우 IP 데이터그램의 크기는 $536 + 40 = 576$ 바이트

TCP 데이터 송수신

bulk 데이터 전송 : ftp, mail, http 등

interactive 데이터 전송 : telnet 등

TCP 데이터 송수신

■ bulk 데이터 전송

- 1000바이트 데이터 송신시
 - 데이터 송신을 위해 write() 또는 send()를 호출
 - TCP는 송신버퍼에 1000바이트를 복사한 후 MSS 단위로 단편화
 - TCP 헤더 20바이트를 붙여 세그먼트로 만들어 IP 계층으로 넘김
 - IP 계층에서는 각 세그먼트에 IP 헤더 20바이트를 붙여 링크 계층으로 보내어 전송함
- 전송한 데이터에 대해 상대방으로부터 Ack가 올 때까지 데이터는 송신버퍼에 보관됨
 - 상대가 Ack를 보내지 않으면 재전송해야하기 때문
- 수신측
 - IP 계층에서 IP 헤더를 제거하고 TCP 계층으로 넘김
 - TCP 계층에서 TCP 헤더를 제거하고 수신버퍼로 데이터를 복사함

Interactive 데이터 송수신

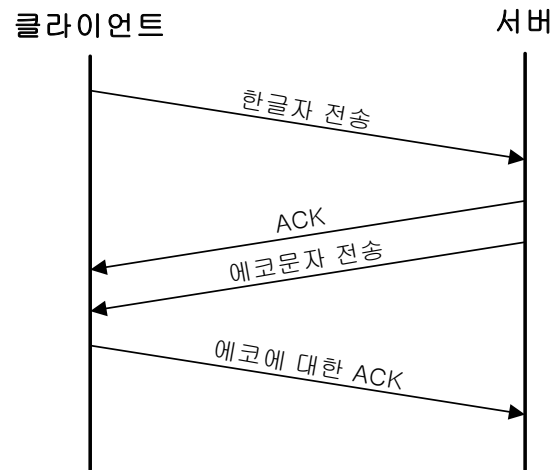


그림 2.12 rlogin에서 한글자 전송 시의 메시지 송수신 절차

Interactive 데이터 송수신

- TCP는 크기가 작은 데이터 송수신 시에 전송 효율이 매우 낮음
 - 예로 rlogin에서 키 입력 시마다 한 바이트씩 데이터를 전송하는데, 글자단위로 에코하므로 한 글자 입력에 4개의 세그먼트 전송이 발생됨
- 응용계층에서는 즉각적인 반응을 원하므로 이러한 비효율을 감수함

흐름제어

수신버퍼에 의한 흐름제어

송신버퍼에 의한 흐름제어

흐름제어

■ 수신버퍼에 의한 흐름제어

- 수신버퍼의 여유 크기를 고려하여 상대방이 전송할 수 있는 데이터량을 바이트 단위로 알려줌
- 상대방이 너무 많은 데이터를 보내지 못하게 하는 흐름제어
- 수신 할수 있는 데이터량을 Window필드(16비트)를 통해 송신측에 알려줌

■ 송신버퍼에 의한 흐름제어

- 송신측에서는 송신버퍼가 부족하면 write(), send()같은 쓰기 함수가 블록됨
- 송신 및 수신 버퍼의 크기 변경을 위해서는 소켓 옵션을 변경
→ TCP 연결이 성립되기 전에 변경해야 함
- 처리 속도가 네트워크 속도에 비해 매우 빠르므로 충분한 메모리가 있음
→ 버퍼 부족으로 인한 흐름제거가 거의 일어나지 않음

혼잡제어(congestion control)

인터넷의 전송 용량 한계로 인해 혼잡이 발생될시 송신측의 전송 속도를 낮춰 혼잡 피함

혼잡제어

- TCP에서는 혼잡제어를 하기 위해 혼잡윈도우(congestion window)를 사용함
- 송신측은 앞에서 설명한 흐름제어윈도우와 혼잡윈도우중 작은 값을 최종 윈도우 크기로 정함
 - 정해진 윈도우 크기 범위 내에서만 송신이 가능함
- 송신측 TCP는 수신측으로부터 정상적으로 Ack가 오면 혼잡윈도우를 1씩 증가시킴
 - 네트워크에 여유가 있다고 판단
- 네트워크에 혼잡이 발생하면 혼잡윈도우 값을 급격히 1로 줄여 전송량을 조정함
- 네트워크에 혼잡이 발생하는 이유
 - 일정한 시간 내에 Ack가 오지 않아 재전송하는 경우
 - 같은 메시지에 대해 Ack를 3회 이상 받는 경우
- 혼잡윈도우 값이 1로 감소했을 경우
 - 정상적으로 송신이 이루어질 때 Ack를 다시 증가시킴
 - 이때 혼잡윈도우 값이 1로 되기 직전의 혼잡윈도우 값의 1/2이 될 때까지 2배씩 증가시킴

2.4 UDP 프로토콜

UDP 헤더

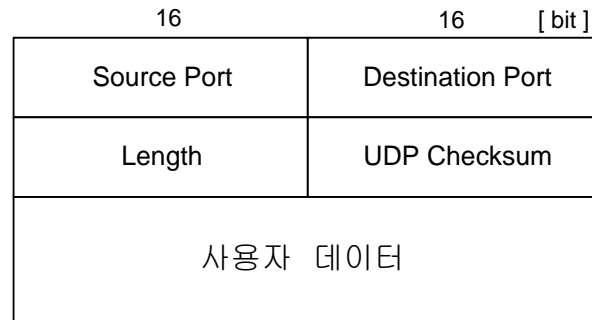


그림 2.13 UDP 프로토콜 데이터 단위

UDP 헤더

- TCP 헤더보다 간단함
- 구성
 - 송신지와 수신지의 포트번호(16비트), 데이터그램 길이(16비트), checksum
 - UDP의 checksum의 적용 범위는 TCP 세그먼트의 경우와 같음
 - 송신지 포트번호는 송신측에서 수행되는 소켓을 구분하기 위한 번호
 - 목적지 포트번호는 목적지 소켓을 구분하기 위한 번호
 - 길이 필드는 헤더와 사용자 데이터를 합한 전체 길이로 16비트임(65535이내 표현)
 - checksum은 헤더와 데이터를 모두 포함한 데이터그램 전체에 대해 에러 탐지에 사용

UDP의 특징

UDP는 비연결형 전송 서비스 제공

UDP 프로토콜의 특징

- 분실 확인이나 전달 순서를 보장하지 않음
 - 연결설정 과정과 종료과정이 없음
 - 스트림을 제공하지 않고 UDP 세그먼트 단위로 송수신이 이루어짐
- TCP에 비하여 헤더의 크기가 작고 연결 지연이 없어 간단한 데이터 전송에 TCP보다 유리
- 흐름제어나 Ack 기능을 제공하지 않으므로 데이터를 전송한 후 버퍼에 남겨두지 않음
- UDP에서도 수신된 세그먼트에서 체크섬으로 에러 검사 후 에러 발생시 데이터 폐기
- UDP를 사용해야 하는 경우
 - 방송 또는 멀티캐스트를 해야 하는 경우
 - 서버 프로그램이 UDP만을 사용하도록 작성되어 있는 경우
 - 오버헤드를 줄이기 위한 실시간 스트리밍 서비스 등

- UDP 소켓의 수신 버퍼의 크기는 리눅스에서 65535로 잡혀있음
 - UDP로 송신 또는 수신할 수 있는 데이터의 크기는 65507(=65535-8-20)
 - 큰 메시지는 MTU 크기 제한에 의해 단편화 되지만 UDP에서는 단편화를 피하는 것이 좋음
→ 전달 순서 확인이나 재전송을 하지 않기 때문

- UDP에서 수신버퍼에 도착한 데이터를 응용 프로그램에서 읽을 때 작은 크기의 버퍼를 사용하면 데이터가 분실될 수 있음
 - UDP에서는 TCP처럼 루프를 돌면서 나머지 데이터를 읽을 수 없음

- UDP 구현에서는 최소한 576 바이트 이상의 수신버퍼를 사용하도록 약속되어 있음
 - UDP 송신 시에 576 바이트 크기 이하의 메시지에 대해서는 수신버퍼 부족현상이 없음

3장

소켓 프로그래밍

- 3.1 소켓의 이해
- 3.2 인터넷 주소변환 체계
- 3.3 TCP 기반 프로그램
- 3.4 UDP 기반 프로그램

Overview

- 트랜스포트 계층 네트워크 프로그래밍 API로 널리 사용되는 소켓에 대한 지식 습득
- 소켓의 정의, 소켓번호와 소켓주소의 개념에 대한 지식 습득
- 소켓을 이용한 네트워크 프로그램 작성법
- 소켓 응용 프로그램으로 에코 서버와 에코 클라이언트 프로그램에 대한 소개

3.1 소켓의 이해

3.1.1 소켓 정의

소켓 정의

TCP나 UDP 같은 전송 계층을 이용하는 API

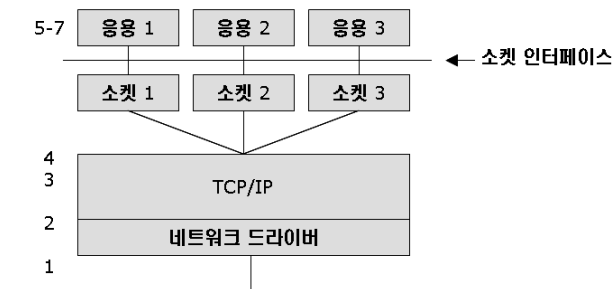


그림 3.1 소켓 인터페이스 위치

소켓 정의

- 1982년 BSD 유닉스 4.1에서 소개
- 모든 유닉스 운영체제에서 제공됨
- 윈도우즈는 윈소크(Winsock)이라는 이름으로 소켓 API를 제공
- 자바 플랫폼에서도 소켓을 이용하기 위한 클래스 제공
- 응용 프로그램에서 TCP/IP를 이용하는 창구 역할을 함
- 소켓 인터페이스
 - 응용 프로그램과 소켓 사이의 인터페이스
- 네트워크 드라이버
 - LAN 카드 같은 네트워크 인터페이스 장치를 구동하는 인터페이스

소켓번호

새로운 소켓을 개설했을 경우 이를 대표하는 int 타입의 번호

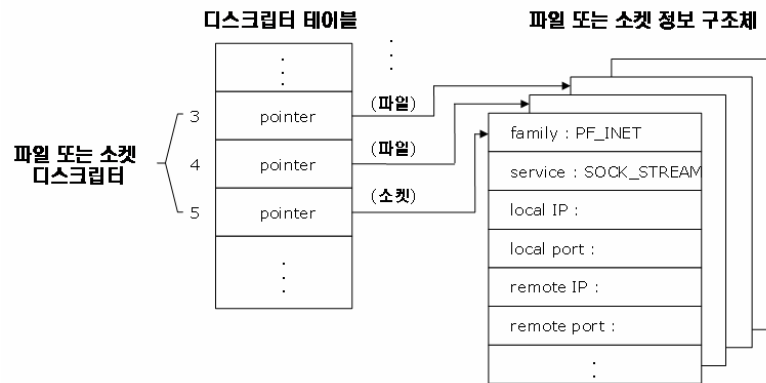


그림 3.2 파일 디스크립터와 소켓 디스크립터

소켓번호

■ 파일 디스크립터

- 유닉스에서 파일을 새로 열면 int 형 타입의 파일 디스크립터(file descriptor)를 리턴
- 프로그램에서 이 파일을 액세스할 때 해당 파일 디스크립터를 사용
- 유닉스에서는 모든 파일, 각종 하드웨어 장치, 파이프, 소켓 등을 파일로 취급
- 파일 디스크립터 테이블
 - 파일을 처음 오픈하면 시스템은 이 파일에 관한 정보를 담고 있는 구조체를 할당
 - 이 구조체를 가리키는 포인터들로 구성된 테이블
 - 이 테이블의 인덱스 값이 파일 디스크립터임

■ 소켓 디스크립터

- 소켓번호라고도 함
- 소켓을 개설하여 얻은 파일 디스크립터
- 응용 프로그램에서 개설된 소켓을 통하여 데이터를 송수신 할 때 소켓 디스크립터 사용

응용 프로그램과 소켓, TCP/IP의 관계

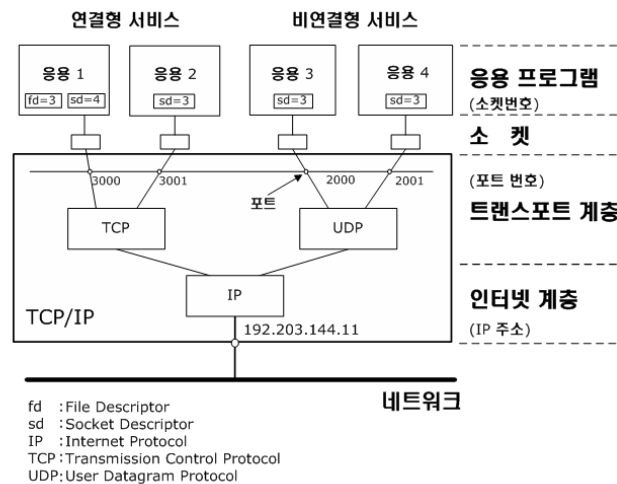


그림 3.3 응용 프로그램과 소켓, TCP/IP의 관계

응용 프로그램과 소켓, TCP/IP의 관계

- 파일 디스크립터 0, 1, 2번
 - 표준 입력(키보드) : 파일 디스크립터 0번
 - 표준 출력 : 파일 디스크립터 1번
 - 표준 에러 : 파일 디스크립터 2번
- 파일 디스크립터 3번 이후
 - 사용자 프로그램에서 파일을 open하여 얻은 파일 디스크립터 배정
- 소켓번호는 응용 프로그램 내에서 순서대로 배정됨
 - 프로그램 내에서만 유일하게 구분
 - 서로 다른 응용 프로그램에서 같은 소켓번호를 사용해도 문제없음

포트번호

IP 데이터그램에 실린 데이터를 최종적으로 전달할 프로세스를 구분
호스트내의 통신 접속점을 구분하기 위해 사용
응용 프로그램을 구분하기 위한 번호

포트번호

- 포트번호의 사용
 - TCP나 UDP 헤더에 실려 16비트로 표현됨
 - TCP나 UDP의 포트번호는 각각 독립적으로 운영되며 1 ~ 65535 사이의 값을 가짐
 - TCP와 UDP가 같은 포트번호를 사용할 수 있음
 - 일반적으로 같은 종류의 서비스를 TCP와 UDP가 동시에 제공할 때만 같은 포트번호 사용

- 미리 정해진 포트번호(well-known 포트)
 - ftp(21번), ssh(22번), telnet(23번), mail(25번) 등
 - 1023번 이하가 배정되며 사용자가 임의로 정하여 사용하는 포트번호는 1024번 이상의 번호임
 - 포트번호는 한 시스템 내에서 멀티캐스트 패킷 수신 등의 특별한 경우를 제외하고는 중복되지 않음

3.1.2 소켓 사용법

소켓 개설

소켓 개설

- 소켓을 개설하여 통신에 이용하기 위해 필요한 정보
 - 통신에 사용할 프로토콜(TCP 또는 UDP)
 - 소켓을 처음 개설할 때 TCP나 UDP의 프로토콜을 선택함
 - 자신의 IP 주소
 - 프로그램이 수행되는 컴퓨터의 IP주소
 - 자신의 포트번호
 - 통신에 사용할 소켓을 구분하는 번호
 - 상대방의 IP 주소
 - 통신하고자 하는 상대방 컴퓨터의 IP 주소
 - 상대방의 포트번호
 - 목적지 컴퓨터 내에서 소켓을 구분하기 위한 포트번호

소켓 프로그래밍

소켓 만들기

socket()의 사용 문법

```
int socket(int protocolFamily,  
           int type,  
           int protocol);
```

소켓 프로그래밍

■ 소켓 만들기

- 소켓 프로그래밍에서 첫 번째로 해야 할 일
- 서버와 클라이언트에서 모두 필요
- socket() 함수를 이용
 - 성공시 새로 만들어진 소켓번호를 리턴
 - 에러 발생시 -1이 리턴되며 전역변수 errno에 에러코드가 들어감

■ socket()의 사용 문법

- protocolFamily(domain)
 - 프로토콜 체계
 - TCP/IP 프로토콜을 사용하기 위해 프로토콜 체계를 인터넷으로 지정하며 PF_INET 선택
 - 그 외에 PF_INET6(IPv6 프로토콜), PF_UNIX(유닉스방식 프로토콜), PF_NS(XEROX 네트워크 시스템의 프로토콜), PF_PACKET(리눅스에서 패킷 캡처를 위해 사용)이 있음

- type
 - 서비스 타입
 - TCP : SOCK_STREAM, UDP : SOCK_DGRAM, Raw : SOCK_RAW 를 각각 선택함
 - Raw 소켓은 TCP나 UDP 계층을 거치지 않고 IP 계층을 바로 이용하는 프로그램에서 사용
- protocol
 - 소켓에서 사용할 프로토콜
 - TCP : IPPROTO_TCP, UDP : IPPROTO_UDP 를 선택
 - type에서 미리 정해진 경우엔 0을 입력

예제) socket() 시스템 콜을 호출하고, 생성된 소켓번호 출력 프로그램

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <sys/socket.h>
6 #include <fcntl.h>
7 #include <unistd.h>
8
9 int main()
10 {
11     int fd1,fd2,sd1,sd2;
12     printf("getdtablesize() = %d\n",getdtablesize());
13
14     fd1 = open("/etc/passwd",O_RDONLY,0);
15     printf("/etc/passwd's file descriptor = %d\n",fd1);
16
17     sd1= socket(PF_INET, SOCK_STREAM,0);
18     printf("stream socket descriptor = %d\n",sd1);
```

- 3 #include <sys/types.h>
 - 소켓 시스템 콜에 필요한 상수 선언
- 4 #include <sys/stat.h>
 - 파일의 상태에 대한 데이터 선언
- 5 #include <sys/socket.h>
 - 소켓 시스템 콜 선언
- 6 #include <fcntl.h>
 - open에 필요한 flag 선언
- 14 fd1 = open("/etc/passwd", O_RDONLY, 0);
 - /etc/passwd 파일을 읽기모드(O_RDONLY)로 열기
- 17 sd1= socket(PF_INET, SOCK_STREAM, 0);
 - 스트림형 소켓 열기


```

19
20     sd2= socket(PF_INET, SOCK_DGRAM, 0);
21     printf("datagram socket descriptor = %d\n",sd2);
22
23     fd2= open("/etc/hosts",O_RDONLY,0);
24     printf("/etc/hosts's file descriptor = %d\n",fd2);
25
26     close(fd2);
27     close(fd1);
28     close(sd2);
29     close(sd1);
30
31     return 0;
32 }

```

- 20 sd2= socket(PF_INET, SOCK_DGRAM, 0);
 - 데이터그램형 소켓 열기

- 23 fd2= open("/etc/hosts",O_RDONLY,0);
 - /etc/hosts 파일을 읽기모드(O_RDONLY)로 열기

- 26 ~ 29번 줄
 - 파일 및 소켓 닫기

실행화면

```
$ socket_number  
getdtablesize() = 1024  
/etc/passwd's file descriptor = 3  
stream socket descriptor = 4  
datagram socket descriptor = 5  
/etc/hosts's file descriptor = 6
```

실행

- 파일명 : socket_number.c
- 컴파일 : gcc -o socket_number socket_number.c
- 사용법 : socket_number

- getdtablesize() = 1024
 - 파일 디스크립터의 최대값은 64 또는 1024임(시스템마다 다름)
 - getdtablesize() 는 한 프로세스에서 개설 가능한 최대 소켓 수를 알려줌

- 파일 디스크립터가 3부터 배정되는 이유
 - 0, 1, 2번은 이미 표준 입 · 출력 및 표준 에러 출력으로 사용됨

소켓주소 구조체

소켓주소는 클라이언트 또는 서버의 구체적인 주소를 표한하기 위해 필요 주소체계(address family), IP 주소, 포트번호로 구성

```
struct sockaddr{
    u_short sa_family;
    char sa_data[14];
};
```

소켓주소 구조체

- 구성
 - 2바이트의 address family
 - u_short는 unsigned short로 types.h 헤더 파일에 정의됨
 - 14바이트의 주소(IP 주소 + 포트번호)

- IP 주소와 포트번호를 구분하여 쓰거나 읽기가 불편함
 - sockaddr_in 구조체를 대신 사용

인터넷 전용 소켓주소

```
struct in_addr{
    u_long s_addr;
};
struct sockaddr_in{
    short    sin_family;
    u_short  sin_port;
    struct    in_addr sin_addr;
    char     sin_zero[8];
};
```

인터넷 전용 소켓주소 (sockaddr_in 구조체)

- 주소체계 sin_family
 - AF_INET (인터넷 주소 체계)
 - socket()으로 소켓을 개설할 때 프로토콜을 PF_INET으로 지정한 소켓에 대한 주소 체계
 - AF_UNIX (유닉스 파일 주소 체계)
 - 값은 2로 셋팅 되어 있음
 - AF_NS (XEROX 주소 체계)
 - 값은 2로 셋팅 되어 있음
- sin_port
 - 16비트의 포트번호
- struct in_addr 구조체
 - 32비트의 IP 주소를 저장할 구조체
- sin_zero[8]
 - 전체 크기를 16바이트로 맞추기 위한 dummy
 - sockaddr 구조체와 호환성을 위해 사용

소켓 사용 절차

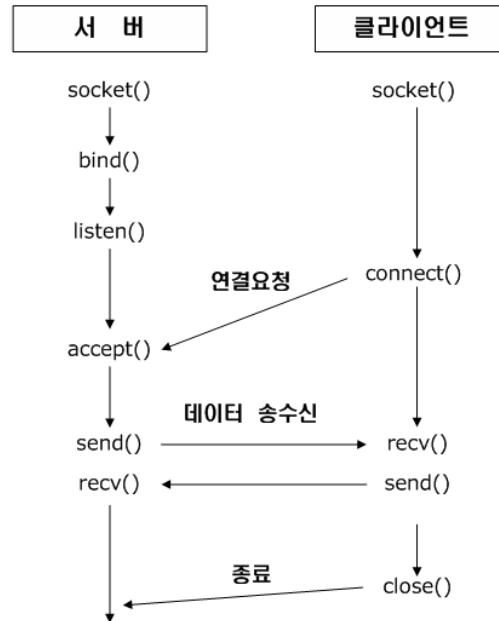


그림 2.4 TCP 소켓 프로그램이 절차

소켓 사용 절차

- 전제 조건
 - 클라이언트-서버 통신 모델에서는 항상 서버 프로그램이 먼저 수행되어야 함
- 서버는 `socket()`을 호출하여 통신에 사용할 소켓을 개설
 - 리턴된 소켓번호와 자신의 소켓주소를 `bind()`를 호출하여 서로 연결
 - `bind()`는 소켓번호는 응용 프로그램 내에서만 알고 있는 통신 창구 번호이고, 소켓주소는 네트워크 시스템만 아는 주소이므로 이들을 묶어야 응용 프로세스와 네트워크 시스템간의 데이터 전달이 가능함
- 서버는 `listen()`을 호출하여 수동 대기모드로 들어감
 - 클라이언트로부터 오는 연결 요청을 처리 가능해짐
 - 서버는 `accept()`함수를 호출하여 클라이언트와 연결설정을 함
 - 클라이언트와 연결이 성공하면 `accept()`가 새로운 소켓을 하나 리턴
 - 이 소켓을 통해 데이터를 송수신함
- 클라이언트는 `socket()`를 호출하여 소켓을 만든 후 서버와 연결설정을 위해 `connect()`를 호출
 - 접속할 상대방 서버의 소켓주소 구조체를 만들어 `connect()`함수의 인자로 줌
 - 클라이언트는 서버와 달리 `bind()`를 호출하지 않음
 - 자신의 IP 주소나 포트번호를 특정한 값으로 지정해 둘 필요가 없음
 - 특정 포트번호를 사용할 땐 `bind()`를 호출함
 - `bind()`를 사용하면 클라이언트 프로그램의 안정성이 떨어짐

3.2 인터넷 주소변환 체계

컴퓨터마다 숫자를 내부에서 표현하는 방식이 다르므로 바이트 순서를 맞춤

3.2.1 바이트 순서

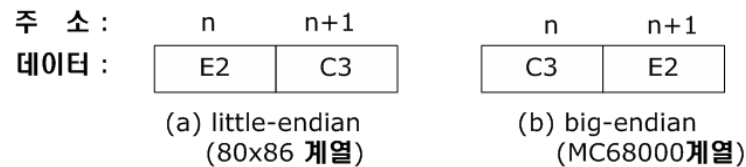


그림 3.5 0xC3E2(십진수 50146)의 호스트 바이트 순서비교

바이트 순서

- 호스트 바이트 순서
 - 컴퓨터가 내부 메모리에 숫자를 저장하는 순서
 - CPU의 종류에 따라 다름
 - little-endian
 - 하위 바이트가 메모리에 먼저 저장됨
 - big-endian
 - 상위 바이트가 메모리에 먼저 저장됨
- 네트워크 바이트 순서
 - 포트번호나 IP 주소와 같은 정보를 바이트 단위로 네트워크로 전송하는 순서
 - high-order(big-endian) 바이트부터 전송하기로 정함
- 바이트 순서가 바뀌는 문제의 해결방법
 - 네트워크로 전송하기 전에 htons() 함수를 사용하여 네트워크 바이트 순서로 바꿈
 - 네트워크로부터 수신한 숫자는 ntohs() 함수를 사용하여 자신의 호스트 바이트 순서로 바꿈
- 바이트 순서를 바꾸는 함수에 변환할 바이트 길이가 2 또는 4바이트일 경우
 - Unsigned short interger 변환(2바이트)
 - htons() : host-to-network 바이트 변환
 - ntohs() : network-to-host 바이트 변환
 - Unsigned long interger 변환(4바이트)
 - htonl() : host-to-network 바이트 변환
 - ntohl() : network-to-host 바이트 변환

바이트 순서 확인

getservbyname() 시스템 콜 함수

```
pmyservent = getservbyname("echo", "udp");
```

servent 구조체

```
struct servent{  
    char *s_name;  
    char **s_aliases;  
    int    s_port;  
    char *s_proto;  
};
```

- getservbyname() 시스템 콜 함수
 - 시스템이 현재 지원하는 특정 응용 프로그램의 정보를 알아내는 함수
 - 서비스 이름과 프로토콜을 인자로 주어 호출
 - 서비스와 관련된 정보를 포함한 servent 라는 구조체의 포인터를 리턴함
- servent 구조체
 - netdb.h 파일에 정의되어 있음
 - 네트워크로부터 받은 정보라 네트워크 바이트 순서로 되어 있음
 - 화면에 출력해 보려면 호스트 바이트 순서로 바꿔야함
 - *s_name : 서비스 이름
 - **s_aliases : 별명 목록
 - s_port : 포트번호
 - *s_proto : 사용하는 프로토콜

바이트 순서 확인 예

```
struct servent *servent
servent = getservbyname("echo", "udp");

if (servent == NULL) {
    printf("서비스 정보를 얻을 수 없음\n\n");
    exit(0);
}

printf("UDP 에코 포트번호(네트워크 순서) : %d\n", servent->s_port);
printf("UDP 에코 포트번호(호스트 순서) : %d\n", ntohs(servent->s_port));
```

실행 결과

```
// 썬 마이크로시스템즈의 sparc system
$ byte_order
UDP 에코 포트번호(네트워크 순서) : 7
UDP 에코 포트번호(호스트 순서) : 7

// 인텔 80x86 계열 PC
$ byte_order
UDP 에코 포트번호(네트워크 순서) : 1792
UDP 에코 포트번호(호스트 순서) : 7
```

실행

- 파일명 : byte_order.c
- 컴파일 : gcc -o byte_order byte_order.c
- 사용법 : byte_order

- 썬 마이크로시스템즈의 Sparc 시스템
 - 호스트 바이트 순서가 네트워크 바이트 순서와 같음

- 인텔 80x86 계열의 CPU 사용 컴퓨터
 - 호스트 바이트 순서가 네트워크 바이트 순서와 다름
 - 1792는 2바이트 숫자 0x0007의 바이트 순서가 서로 바뀐 0x07

3.2.2 IP 주소변환

4바이트의 IP 주소를 편의에 따라 도메인 네임 또는 dotted decimal 방식으로 표현함

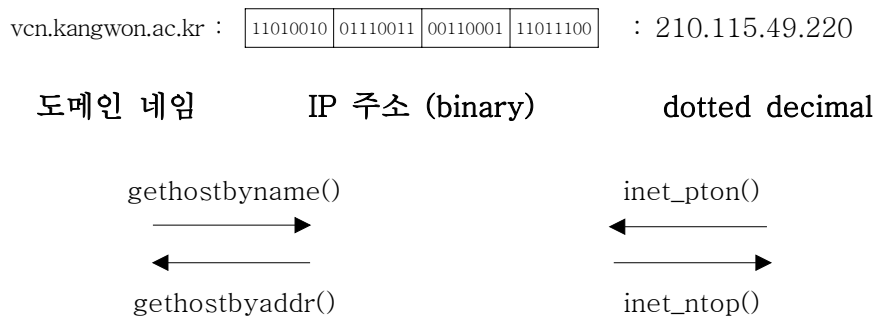


그림 3.6 IP 주소 표현의 세 가지 방법 및 이들의 상호 변환 함수

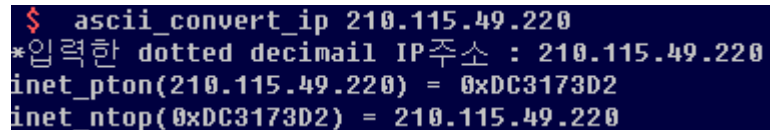
- IP 주소
 - dotted decimal 표현의 저장에는 15개의 문자로 구성된 스트링 변수가 사용됨
 - IP 데이터그램을 네트워크로 실제로 전송할 때 IP 헤더에는 4바이트의 binary IP 주소만 사용
- gethostbyname()
 - 도메인 네임을 binary IP 주소로 변환해주는 함수
- gethostbyaddr()
 - binary IP 주소를 도메인 네임으로 변환해주는 함수
- inet_pton()
 - dotted decimal 주소를 binary IP 주소로 변환해주는 함수
- inet_ntop()
 - binary IP 주소를 dotted decimal 주소로 변환해주는 함수

ASCII(dotted decimal)로 표현된 주소를 4바이트 IP 주소로 변환

```
inet_pton(AF_INET, argv[1], &inaddr.s_addr);
printf("inet_pton(%s) = 0x%X \Wn", argv[1], inaddr.s_addr);

inet_ntop(AF_INET, &inaddr.s_addr, buf, sizeof(buf));
printf("inet_ntop(0x%X) = %s \Wn", inaddr.s_addr, buf);
```

실행화면



```
$ ascii_convert_ip 210.115.49.220
*입력한 dotted decimal IP주소 : 210.115.49.220
inet_pton(210.115.49.220) = 0xDC3173D2
inet_ntop(0xDC3173D2) = 210.115.49.220
```

ASCII(dotted decimal)로 표현된 주소를 4바이트 IP 주소로 변환

- inet_pton(AF_INET, argv[1], &inaddr.s_addr)
 - dotted decimal로 표현된 주소를 명령문 인자로 입력
 - 4바이트의 IP주소로 바꿈

- inet_ntop(AF_INET, &inaddr.s_addr, buf, sizeof(buf));
 - IP 주소로부터 dotted decimal 주소를 얻음

- 실행결과
 - 파일명 : ascii_convert_ip.c
 - 컴파일 : gcc -o ascii_convert_ip ascii_convert_ip.c
 - 사용법 : ascii_convert_ip 210.115.49.220

도메인 주소변환

DNS 서버의 도움을 받아 도메인 네임으로부터 IP 주소를 얻거나 IP주소로부터 도메인 네임 얻음

```
#include<netdb.h>
struct hostent *gethostbyname(const char *hname);
struct hostent *gethostbyaddr(const char *in_addr, int len, int family);
```

■ 도메인 주소변환 함수

- gethostbyname
 - 도메인 네임 hname를 스트링 형태로 입력
 - 이에 해당하는 호스트의 정보를 가진 hostent 구조체 포인터를 리턴
- gethostbyaddr
 - IP 주소를 포함하고 있는 구조체 in_addr의 포인터와 주소의 길이, 주소 타입을 입력
 - 해당 호스트의 정보를 가진 hostent 구조체 포인터를 리턴

■ IP 주소로부터 호스트 정보를 얻는 방법

- DNS서버에서 Reverse DNS가 수행되어야함
 - 최근 보안문제로 대부분의 DNS 서버가 제공하고 있지 않음

구조체 hostent의 정의

hostent 포인터

호스트의 각종 정보를 저장한 구조체

```
struct hostent {  
    char *h_name  
    char **h_aliases  
    int  h_addrtype  
    int  h_length  
    char **h_addr_list  
};
```

구조체 hostent의 정의

- char *h_name
 - 호스트 이름
- char **h_aliases
 - 호스트 별명들
- int h_addrtype
 - 호스트 주소의 종류
- int h_length
 - 주소의 크기
- char **h_addr_list
 - IP 주소 리스트

도메인 이름을 IP주소로 변환 예제

```
struct hostent *hp;
struct in_addr in;

hp = gethostbyname(argv[1]);
if (hp == NULL) {
    printf("gethostbyname fail Wn");
    exit(0);
}

printf("호스트 이름 : %s Wn", hp->hname);
printf("호스트 주소타입 번호 : %d Wn", hp->h_addrtype);
printf("호스트 주소의 길이 : %d Wn", hp->h_length);

for(i=0; hp->h_addr_list[i]; i++) {
    memcpy(&in.s_addr, hp->h_addr_list[i], sizeof(in.s_addr));
    inet_ntop(AF_INET, &in, buf, sizeof(buf));
    printf("IP주소(%d번째) : %s Wn", i+1, buf);
}

for (i=0; hp->h_aliases[i]; i++)
    printf("호스트 별명(%d 번째) : %s", i+1, hp->h_aliases[i]);
```

실행 화면

```
$ get_hostinfo vcn.kangwon.ac.kr
호스트 이름      : vcn.kangwon.ac.kr
호스트 주소타입 번호 : 2
호스트 주소의 길이  : 4바이트
IP주소(1 번째)   : 210.115.36.127
호스트 별명(1 번째) : vcn
```

실행

- 파일명 : get_hostinfo.c
- 컴파일 : gcc -o get_hostinfo get_hostinfo.c
- 사용법 : get_hostinfo vcn.kangwon.ac.kr

gethostbyaddr() 사용 예제

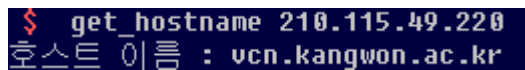
```
struct hostent *myhost;
struct in_addr in;

inet_pton(AF_INET, argv[1], &in.s_addr);
myhost = gethostbyaddr((char *)&in.s_addr, sizeof(in.s_addr), AF_INET);

if (myhost == NULL) {
    printf("Error at gethostbyaddr() Wn");
    exit(0);
}

printf("호스트 이름 : %s Wn", myhost->h_name);
```

실행화면



```
$ get_hostname 210.115.49.220
호스트 이름 : vcn.kangwon.ac.kr
```

실행

- 파일명 : get_hostname.c
- 컴파일 : gcc -o get_hostname get_hostname.c
- 사용법 : get_hostname 210.115.49.220

3.3 TCP 기반 프로그램

3.3.1 TCP 클라이언트 프로그램

TCP 클라이언트 프로그램 작성 절차

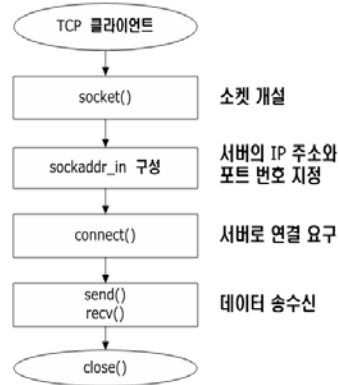


그림 3.7 TCP 클라이언트 프로그램 작성 절차

TCP 클라이언트 프로그램 작성 절차

- socket(), 소켓 개설
 - 클라이언트는 먼저 socket()으로 소켓을 개설
 - TCP 또는 UDP 소켓을 선택하며, TCP의 경우 서비스 type 인자를 SOCK_STREAM으로 선택
 - 소켓을 이용한 통신 프로그램에서는 사용할 전송 프로토콜, 자신의 IP 주소와 프로토콜, 상대방 IP 주소와 포트번호 지정되어야 함
 - 소켓을 만들 때는 전송 프로토콜만을 지정하게 됨
 - socket() 수행시 내부적으로 일어나는 동작

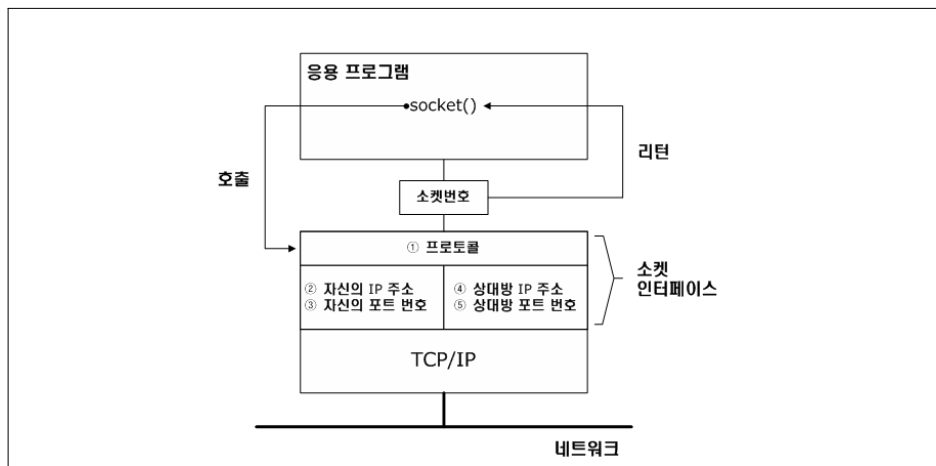


그림 3.8 socket() 호출시 소켓번호와 소켓 인터페이스의 관계

connect(), 서버에 연결 요청

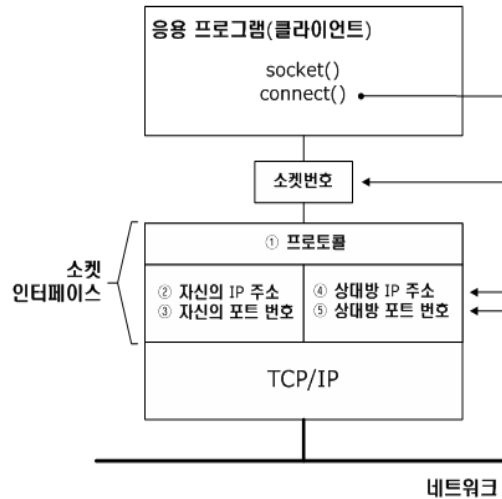


그림 3.9 connect() 호출시 소켓번호와 소켓주소의 관계

■ connect(), 서버에 연결 요청

- connect()를 호출하여 서버에게 연결 요청
- connect() 사용 문법

```
int connect(int s, const struct sockaddr *addr, int addrlen);
```

→ int s : 소켓번호

→ const struct sockaddr *addr : 상대방 서버의 소켓주소 구조체

→ int addrlen : 구조체 *addr의 크기

- connect()의 수행 내용

→ 3-way 핸드셰이크가 성공하여 서버와 연결되면 connect()는 0을 리턴

→ 실패시 -1을 리턴하며 전역변수 errno에 에러코드가 들어감

→ 서버와 연결이 되려면 서버측은 listen()과 accept()를 호출해 두어야함

- connect() 호출 중 에러 발생시

→ 해당 소켓을 close()로 닫고 새로운 소켓을 만든 후 사용

데이터 송수신

문법	인자	
int send(int s, char* buf, int length, int flags);	s	소켓번호
	buf	전송할 데이터가 저장된 버퍼
	length	buf 버퍼의 크기
	flags	보통 0
int write(int s, const void* buf, int length);	s	소켓번호
	buf	전송할 데이터가 저장된 버퍼
	length	buf 버퍼의 길이
int recv(int s, char* buf, int length, int flags);	s	소켓번호
	buf	수신 데이터가 저장된 버퍼
	length	buf 버퍼의 크기
	flags	보통 0
int read(int s, const void* buf, int length);	s	소켓번호
	buf	수신 데이터가 저장된 버퍼
	length	buf 버퍼의 길이

표 3.1 TCP 소켓의 데이터 송수신 함수

■ 데이터 송수신

- 클라이언트가 서버와 연결되면 send(), recv() 또는 write(), read()를 사용하여 데이터 송수신
- send(), write()
 - 스트림형 소켓을 통하여 데이터를 송신하는 함수
 - 데이터를 전송할 소켓번호(s), 송신할 데이터 버퍼(buf), 전송할 데이터 크기(length)를 지정
 - send()는 flags 인자를 추가로 사용가능함
 - 전송된 데이터 크기를 바이트 단위로 리턴함
- recv(), read()
 - 스트림형 소켓을 통하여 데이터를 수신하는 함수
 - 데이터를 수신할 소켓번호(s), 송신할 데이터 버퍼(buf), 전송할 데이터 크기(length)를 지정
 - 읽은 데이터 크기를 바이트 단위로 리턴함
- send(), recv() 함수는 flags 옵션을 지정할 수 있는데 일반 데이터 송수신 시에는 0으로 선택
 - flags는 데이터 송수신시 특별한 옵션을 지정할 때 사용

- MSS보다 큰 데이터를 send()나 write()로 송신 시
 - 전체 데이터가 MSS 크기로 분할되어 전송됨
 - TCP에서는 세그먼트 순서 확인과 데이터그램 분실을 수신측에서 검사 및 재전송

- TCP 소켓에서 write()나 send() 실행 시
 - 데이터가 TCP 계층에 있는 송신버퍼로 들어감
 - 송신 버퍼가 비어 있지 않아 데이터를 이곳에 쓸 수 없으면 프로세스는 블록 상태로 감
 - 소켓 모드가 블록형일 때 프로그램은 write()문에서 기다림
 - write()문이 블록된 경우 송신버퍼에 먼저 들어가 있던 데이터가 전송
 - write한 데이터가 송신버퍼로 모두 이동되면 write()문이 리턴
 - write()문이 리턴되었다는 것은 데이터가 자신의 TCP에 있는 송신버퍼에 들어감을 의미

close(), 소켓 닫기

소켓의 사용을 마치고 소켓을 닫을 때 사용

■ close(), 소켓 닫기

- 해당 소켓번호를 지정하여 close()를 호출하여 소켓을 닫음
- 클라이언트나 서버 중 누구나 먼저 호출 가능
- close()를 호출한 시점에서 송신버퍼에 있으나 아직 전송되지 않거나 네트워크에 전달중인 데이터가 있을 경우
 - 모든 데이터의 전달 후 TCP 연결이 종료됨
 - 소켓 옵션을 변경하여 미전송된 데이터를 버리고 종료 할 수도 있음

3.3.2 TCP 클라이언트 예제 프로그램

daytime 클라이언트

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5
6 #define BUF_LEN 128
7
8 int main(int argc, char *argv[])
9 {
10     int s, n;
11     char *haddr;
12     struct sockaddr_in server_addr;
13     char buf[BUF_LEN+ 1];
14
15     if(argc != 2) {
16         printf("usage: %s ip_address\n", argv[0]);
17         exit(0);
18     }
19     haddr = argv[1];
20
21     if((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
22         printf("can't create socket\n");
23         exit(0);
24     }
```

- 12 struct sockaddr_in server_addr;
 - 서버의 소켓주소 구조체
- 21 if((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
 - 소켓 생성

```

25
26     bzero((char *)&server_addr, sizeof(server_addr));
27
28     server_addr.sin_family = AF_INET;
29     server_addr.sin_addr.s_addr = inet_addr(argv[1]);
30     server_addr.sin_port = htons(13);
31
32     if(connect(s, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
33         printf("can't connect.\n");
34         exit(0);
35     }
36
37     while((n = read(s, buf, BUF_LEN)) > 0) {
38         buf[n] = '\0';
39         printf("%s", buf);
40     }
41
42     close(s);
43
44     return 0;
45 }

```

- 26 `bzero((char *)&server_addr, sizeof(server_addr));`
 - 서버의 소켓주소 구조체 `servaddr`을 'W0'으로 초기화
- 28 `server_addr.sin_family = AF_INET;`
 - 주소 체계 선택
- 29 `server_addr.sin_addr.s_addr = inet_addr(argv[1]);`
 - 32비트의 IP주소로 변환
- 30 `server_addr.sin_port = htons(13);`
 - daytime 서비스 포트 번호
- 32 `if(connect(s, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {`
 - 연결 요청
- 37 `while((n = read(s, buf, BUF_LEN)) > 0) {(n = read(s, buf, BUF_LEN)`
 - 서버가 보내오는 daytime 데이터의 수신

실행화면

```
$ tcp_daytime 210.115.49.220  
Mon Sep 12 17:42:06 2005
```

- 파일명 : tcp_daytime.c
- 컴파일 : gcc -o tcp_daytime tcp_daytime.c
- 사용법 : tcp_daytime 210.115.49.220

TCP 에코 클라이언트

well-known 포트 7번으로 제공되며 클라이언트가 보낸 문자열을 다시 클라이언트로 전송
에코 서비스를 요청하는 TCP 클라이언트

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <unistd.h>
6
7 #define BUF_LEN 128
8
9 int main(int argc, char *argv[])
10 {
11     int s, n, len_in, len_out;
12     struct sockaddr_in server_addr;
13     char *haddr;
14     char buf[BUF_LEN+1];
15
16     if(argc != 2) {
17         printf("usage: %s ip_address Wn", argv[0]);
18         exit(0);
19     }
20
21     haddr = argv[1];
22
23     if((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
24         printf("can't create socket Wn");
25         exit(0);
26     }
27
28     bzero((char *)&server_addr, sizeof(server_addr));
29
```

에코 서비스를 요청하는 TCP 클라이언트

- 28 bzero((char *)&server_addr, sizeof(server_addr));
- 에코 서버의 소켓주소 구조체 작성


```

30  server_addr.sin_family = AF_INET;
31  server_addr.sin_addr.s_addr = inet_addr(haddr);
32  server_addr.sin_port = htons(7);
33
34  if(connect(s, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
35      printf("can't connect. Wn");
36      exit(0);
37  }
38
39  printf("Input any string : ");
40  if (fgets(buf, BUF_LEN, stdin))
41  {
42      buf[BUF_LEN] = '\0';
43      len_out = strlen(buf);
44  } else {
45      printf("fgets error Wn");
46      exit(0);
47  }
48  if (write(s, buf, len_out) < 0) {
49      printf("write error Wn");
50      exit(0);
51  }
52  printf("Echoed string : ");
53  for(len_in=0; n = 0; len_in < len_out; len_in += n) {
54      if((n = read(s, &buf[len_in], len_out - len_in)) < 0) {
55          printf("read error Wn");
56          exit(0);
57      }
58  }
59  printf("%s", buf);
60  close(s);
61 }

```

- 30 ~ 32번 줄
 - echo 서버의 소켓주소 구조체 작성
- 34 if(connect(s, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
 - 연결 요청
- 40 if (fgets(buf, BUF_LEN, stdin))
 - 키보드로 입력받음
- 62 if (write(s, buf, len_out) < 0) {
 - echo 서버로 메시지 송신
- 73 printf("%s", buf);
 - 수신된 echo 메시지를 화면 출력

실행 화면

```
$ tcp_echocli 210.115.49.220
Input any string : tcp_echocli test
Echoed string : tcp_echocli test
```

실행

- 파일명 : tcp_echocli.c
- 컴파일 : gcc -o tcp_echocli tcp_echocli.c
- 사용법 : tcp_echocli 210.115.49.220

3.3.3 TCP 서버 프로그램

TCP 서버 프로그램 작성절차

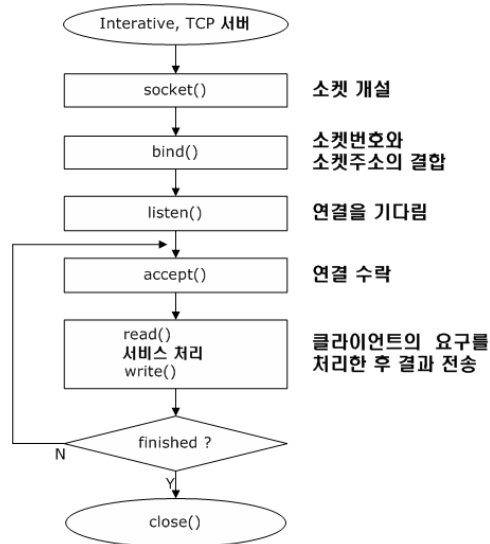


그림 3.10 Iterative 모델의 TCP 서버

TCP 서버 프로그램 작성절차

- socket(), 소켓의 생성
 - 서버도 클라이언트와 통신을 하기 위해 소켓을 생성

```
socket(PF_INET, SOCK_STREAM, 0);
```

bind - 소켓번호와 소켓주소를 연결함

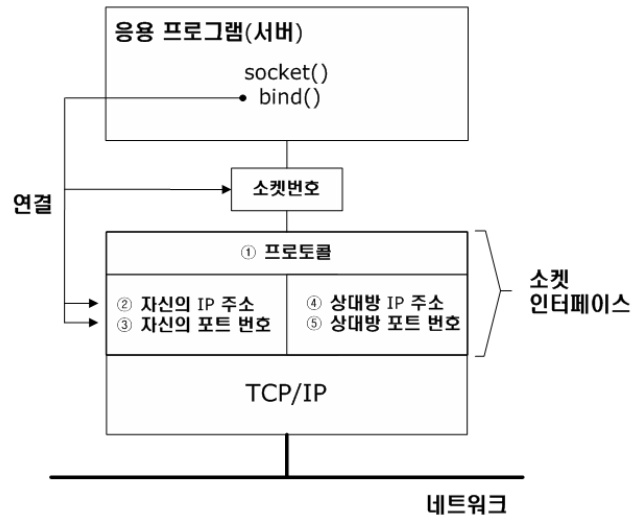


그림 3.11 bind() 호출시 소켓번호와 소켓주소의 관계

■ bind

- socket()으로 생성된 소켓의 소켓번호는 응용 프로그램만 알고 있음
→ 컴퓨터 외부와 통신하기 위해 소켓번호와 소켓주소를 연결해 두어야함
- bind()가 필요한 이유
→ 임의의 클라이언트가 서버 프로그램의 특정 소켓으로 접속을 하려면 서버는 자신의 소켓번호와 클라이언트가 알고 있는 자신의 IP 주소 및 포트번호를 미리 연결해 두어야함
- bind() 사용 문법

```
int bind(int s, struct sockaddr *addr, int len);
```

- int s : 소켓 번호
- struct sockaddr *addr : 서버 자신의 소켓주소 구조체 포인터
- int len : *addr 구조체의 크기

- bind()는 성공시 0, 실패시 -1을 리턴

bind() 사용 예

```
#define SERV_IP_ADDR "210.115.49.220"
#define SERV_PORT 5000

s=socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr_in server_addr;

server_addr.sin_family=AF_INET;
server_addr.sin_addr.s_addr=inet_addr(SERV_IP_ADDR);
server_addr.sin_port=htons(SERV_PORT);

bind(s, (struct sockaddr *)&server_addr, sizeof(server_addr));
```

bind() 사용 예

■ 소켓 생성

- s=socket(PF_INET, SOCK_STREAM, 0);
- struct sockaddr_in server_addr;

■ 소켓 구조체 내용

- server_addr.sin_family=AF_INET;
- server_addr.sin_addr.s_addr=inet_addr(SERV_IP_ADDR);
- server_addr.sin_port=htons(SERV_PORT);

■ 소켓번호와 소켓 주소를 bind

- bind(s, (struct sockaddr *)&server_addr, sizeof(server_addr));

■ INADDR_ANY

- 서버가 자신의 IP 주소를 자동으로 가져다 쓸 때 사용

```
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

listen()

사용 문법

```
int listen(int s, int backlog);
```

int s : 소켓번호

int backlog : 연결을 기다리는 클라이언트의 최대 수

listen()

■ 동작 순서

- 클라이언트가 listen()을 호출해 둔 서버 소켓을 목적지로 connect()를 호출
→ 3-way 핸드셰이크 연결설정의 시작
- 시스템이 핸드셰이크를 마친 후에는 서버 애플리케이션이 설정된 연결을 받아들이는 과정
→ accept()가 사용됨
- accept()는 한번에 하나의 연결만 가져감
→ 여러 연결요청이 동시에 오면 시스템은 설정된 연결들을 accept 큐에 넣고 대기함
→ backlog 인자는 대기시킬 수 있는 연결의 최대 수

■ listen은 소켓을 단지 수동 대기모드로 바꾸어 주는 것

- 성공시 0, 실패시 -1을 리턴

accept()

서버가 listen()의 호출 이후 클라이언트와 설정된 연결을 실제로 받기 위해 사용

사용 문법

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

int s : 소켓번호

struct sockaddr *addr : 연결요청을 한 클라이언트의 소켓주소 구조체

int *addrlen : *addr 구조체 크기의 포인터

accept()

■ 특징

- 수행 성공시 클라이언트와의 통신에 사용할 새로운 소켓이 생성됨
→ 실패시에는 -1이 리턴
- 서버는 클라이언트와 통신하기 위해 새로 만들어진 소켓번호를 사용함
- 연결된 클라이언트의 소켓주소 구조체와 소켓주소 구조체의 길이의 포인터를 리턴
→ addr 과 addrlen 인자로 리턴함
→ 서버는 addr 소켓주소 내용으로 연결된 클라이언트의 IP 주소를 알 수 있음

3.3.4 TCP 에코 서버 프로그램

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4
5 #define BUF_LEN 128
6
7 int main(int argc, char *argv[])
8 {
9     struct sockaddr_in server_addr, client_addr;
10    int server_fd, client_fd;
11    int len, msg_size;
12    char buf[BUF_LEN+1];
13
14    if(argc != 2)
15    {
16        printf("usage: %s portWn", argv[0]);
17        exit(0);
18    }
19
```

- 10 int server_fd, client_fd;
- 소켓 번호


```

20  if((server_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
21  {
22      printf("Server: Can't open stream socket.");
23      exit(0);
24  }
25
26  bzero((char *)&server_addr, sizeof(server_addr));
27
28  server_addr.sin_family = AF_INET;
29  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
30  server_addr.sin_port = htons(atoi(argv[1]));
31
32  if(bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
33  {
34      printf("Server: Can't bind local address.\n");
35      exit(0);
36  }
37
38  listen(server_fd, 5);
39
40  while(1)
41  {
42      printf("Server : waiting connection request.\n");
43      len = sizeof(client_addr);
44

```

- 20 if((server_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
 - 소켓 생성
- 26 bzero((char *)&server_addr, sizeof(server_addr));
 - server_addr 을 '0' 으로 초기화
- 28 ~ 30번 줄
 - server_addr 셋팅
- 32 if(bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
 - bind() 호출
- 38 listen(server_fd, 5);
 - 소켓을 수동 대기모드로 셋팅

```

45     client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &len);
46     if(client_fd < 0)
47     {
48         printf("Server: accept failed.\n");
49         exit(0);
50     }
51
52     printf("Server : A client connected.\n");
53     msg_size = read(client_fd, buf, sizeof(buf));
54     write(client_fd, buf, msg_size);
55     close(client_fd);
56 }
57
58 close(server_fd);
59 return 0;
60 }

```

■ 40 ~ 56번 줄

- iterative echo 서비스 수행

■ 45 client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &len);

- 연결요청을 기다림

실행화면

```
$ tcp_echoserv 5555  
Server : waiting connection request.  
Server : A client connected.  
Server : waiting connection request.
```

실행

- 파일명 : tcp_echoserv.c
- 컴파일 : gcc -o tcp_echoserv tcp_echoserv.c
- 사용법 : tcp_echoserv 포트번호

3.4 UDP 프로그램

3.4.1 UDP 프로그램 작성 절차

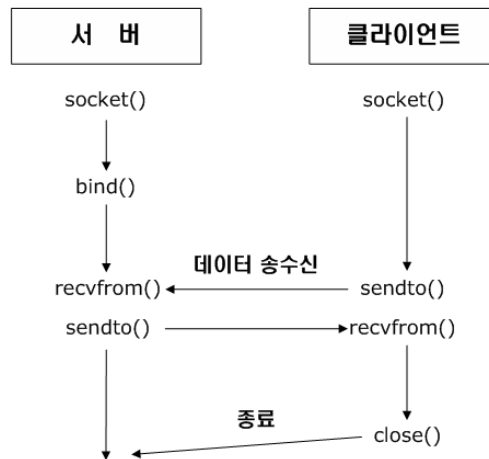


그림 3.12 UDP 소켓 프로그래밍 절차

UDP 프로그램 작성 절차

■ 특징

- type 인자로 `SOCK_DGRAM`을 지정
- UDP 소켓은 특정 호스트와의 일 대 일 통신이 아니라 임의의 호스트와 데이터그램 송수신
- 비연결형 소켓
 - 연결설정을 위한 `connect()` 시스템 콜을 사용할 필요 없음
 - 소켓 개설 후 바로 임의의 상대방과 데이터 송수신 가능
- 데이터 송수신시 각 데이터그램마다 목적지의 IP주소와 포트번호를 항상 함수 인자로 줌

sendto()와 recvfrom() 함수의 사용법

문법	인자	
int sendto(int s, char* buf, int length, int flags, sockaddr* to, int tolen)	s	소켓번호
	buf	전송할 데이터가 저장된 버퍼
	length	buf 버퍼의 크기
	flags	보통 0
	to	목적지의 소켓주소 구조체
	tolen	to 버퍼의 크기
int recvfrom(int s, char* buf, int length, int flags, sockaddr* from, int* fromlen)	s	소켓번호
	buf	수신할 데이터가 저장된 버퍼
	length	buf 버퍼의 크기
	flags	보통 0
	from	발신자의 소켓주소 구조체
	fromlen	from 버퍼의 크기

표 3.2 sendto()와 recvfrom() 함수의 사용법

sendto()와 recvfrom() 함수의 사용법

■ sendto()

- UDP 소켓을 통한 데이터의 송신 함수

■ recvfrom()

- UDP 소켓을 통한 데이터의 수신 함수
- 성공적으로 수행되면 from 구조체에 데이터그램을 보낸 상대방의 소켓주소가 들어감
- 데이터를 보낸 발신자에게 데이터를 보낼 때
→ from에 있는 소켓주소를 sendto()의 to로 복사하여 사용
- fromlen에는 from 구조체의 길이가 정수형 포인터로 리턴됨

3.4.2 UDP 에코 프로그램

UDP 에코 클라이언트

echo 서비스를 수행하는 UDP 클라이언트

```
1 #include <stdio.h>
2 #include <sys/socket.h>
3 #include <arpa/inet.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <unistd.h>
7
8 #define ECHOMAX 255
9
10 int main(int argc, char *argv[])
11 {
12     int sock;
13     struct sockaddr_in echoServAddr;
14     struct sockaddr_in fromAddr;
15     unsigned short echoServPort;
16     unsigned int fromSize;
17     char *servIP;
18     char *echoString;
19     char echoBuffer[ECHOMAX+1];
20     int echoStringLen;
21 }
```

- 12 ~ 20번 줄
 - 프로그램에 사용될 변수 선언

```

32     if((echoStringLen=strlen(echoString))>ECHOMAX)
33     {
34         printf("Echo word too long");
35         exit(0);
36     }
37
38     if(argc=4)
39         echoServPort=atoi(argv[3]);
40     else
41         echoServPort=7;
42
43     if((sock=socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP))<0)
44     {
45         printf("socket() failed");
46         exit(0);
47     }
48
49     memset(&echoServAddr, 0, sizeof(echoServAddr));
50     echoServAddr.sin_family=AF_INET;
51     echoServAddr.sin_addr.s_addr=inet_addr(servIP);
52     echoServAddr.sin_port=htons(echoServPort);
53
54     if(sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *)&echoServAddr,
55             sizeof(echoServAddr))!=echoStringLen){
56         printf("sendto() sent a different number of bytes than expected");
57         exit(0);
58     }
59

```

- 38 ~ 41번 줄
 - 에코서버 포트번호를 받음
 - 잘못된 값 입력시 디폴트 7로 설정
- 43 ~ 47번 줄
 - 소켓 생성
- 49 ~ 52번 줄
 - 에코 서버의 소켓주소 구조체 작성
- 54 ~ 58번 줄
 - 에코 서버로 메시지 송신

```

60     fromSize=sizeof(fromAddr);
61
62     if((respStringLen=recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *)&fromAddr,
63         &fromSize))!=echoStringLen){
64         printf("recvfrom() failed");
65         exit(0);
66     }
67
68     if(echoServAddr.sin_addr.s_addr !=fromAddr.sin_addr.s_addr)
69     {
70         fprintf(stderr,"Error: received a packet from unknown source.\n");
71         exit(1);
72     }
73
74     echoBuffer[respStringLen]='\0';
75     printf("Received: %s\n", echoBuffer);
76     close(sock);
77     exit(0);
78 }

```

- 62 ~ 66번 줄
 - 에코 메시지 수신

- 75 printf("Received: %s\n", echoBuffer);
 - 수신된 메시지 출력

실행화면

```
$ udp_echocli 210.115.49.220 test_message 6000  
Received: test_message
```

실행

- 파일명 : udp_echocli.c
- 컴파일 : gcc -o udp_echocli udp_echocli.c
- 사용법 : udp_echocli 210.115.49.220 "Message" 포트번호

UDP 에코 서버

echo 서비스를 수행하는 UDP 서버

```
1 #include <stdio.h>
2 #include <sys/socket.h>
3 #include <arpa/inet.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <unistd.h>
7
8 #define ECHOMAX 255
9
10 int main(int argc, char *argv[])
11 {
12     int sock;
13     struct sockaddr_in echoServAddr;
14     struct sockaddr_in echoClntAddr;
15     unsigned int cliAddrLen;
16     char echoBuffer[ECHOMAX];
17     unsigned short echoServPort;
18     int recvMsgSize;
19
```

- 12 ~ 18번 줄
 - 프로그램에 사용될 변수 선언

```

20     if(argc !=2)
21     {
22         fprintf(stderr, "Usage: %s<UDP SERVER PORT>Wn", argv[0]);
23         exit(1);
24     }
25
26     echoServPort=atoi(argv[1]);
27
28     if((sock=socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP))<0)
29     {
30         printf("socket() failed");
31         exit(0);
32     }
33
34     memset(&echoServAddr, 0, sizeof(echoServAddr));
35     echoServAddr.sin_family=AF_INET;
36     echoServAddr.sin_addr.s_addr=htonl(INADDR_ANY);
37     echoServAddr.sin_port=htons(echoServPort);
38
39     if(bind(sock, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr)) <0)
40     {
41         printf("bind() failed");
42         exit(0);
43     }
44

```

■ 28 ~ 32번 줄

- 소켓 생성

■ 34 ~ 37번 줄

- 에코 서버의 소켓주소 구조체 작성

■ 39 ~ 43번 줄

- bind() 호출

```

45     for(;;)
46     {
47         cliAddrLen=sizeof(echoClntAddr);
48
49         if((recvMsgSize=recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *)&echoClntAddr,
                                &cliAddrLen))<0){
50             printf("recvfrom() failed");
51             exit(0);
52         }
53
54         printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));
55
56         if(sendto(sock, echoBuffer, recvMsgSize, 0, (struct sockaddr *)&echoClntAddr,
                    sizeof(echoClntAddr)) !=recvMsgSize)
57         {
58             printf("sendto() sent a different number of bytes than expected");
59             exit(0);
60         }
61     }
62 }

```

- 45 ~ 61번 줄
 - 에코 서비스 수행
- 49 ~ 52번 줄
 - 메시지 수신

실행화면

```
$ udp_echoerv 6000  
Handling client 210.115.49.220
```

실행

- 파일명 : udp_echoerv.c
- 컴파일 : gcc -o udp_echoerv udp_echoerv.c
- 사용법 : udp_echoerv 포트번호

4장

고급 소켓 프로그래밍

- 4.1 소켓의 동작 모드
- 4.2 다중처리 기술
- 4.3 비동기형 채팅 프로그램
- 4.4 폴링형 채팅 프로그램

Overview

- 복잡한 통신 프로그램을 위한 다중처리 기술 습득

4.1 소켓의 동작 모드

블록(blocking) 모드

넌블록(non-blocking) 모드

비동기(asynchronous) 모드

블록 모드

- 소켓에 대해 시스템 콜 호출시 시스템이 동작 완료할 때까지 시스템 콜에서 프로세스가 멈추어 있는 모드
- 소켓을 처음 생성할 때 디폴트로 블록 모드가 됨
- 블록 모드의 소켓을 사용하는 프로그램에서 프로세스가 영원히 블록 상태가 될 수 있음
- 소켓 관련 시스템 콜 중에 블록될 수 있는 예
 - listen(), connect(), accept(), recv(), send(), read(), write(), recvfrom(), sendto(), close()
- 응용 프로그램이 일 대 일 통신을 하거나 한가지 작업만 할 경우에 사용

넌블록 모드

- 소켓 관련 시스템 콜에 대하여 시스템이 즉시 처리할 수 있으면 바로 결과를 리턴
- 즉시 처리할 수 없는 경우 시스템 콜이 바로 리턴되어 응용프로그램이 블록되지 않음
- 폴링을 주로 사용
 - 다중화를 위해 시스템 콜이 성공적으로 실행될 때까지 계속확인 하는 방법
- 통신상대가 여럿이거나 여러 작업을 병행하여 실행할 경우 사용

비동기 모드

- 소켓에서 어떤 I/O변화 발생시 그 사실을 응용 프로그램이 알 수 있도록 함
 - 이 때 입출력 처리 등 원하는 동작을 함
- 소켓을 비동기로 바꾸는 방법
 - select() 함수를 이용
 - I/O변화가 발생할 수 있는 소켓들 전체를 대상으로 select()를 호출해둠
 - 이 중 임의의 소켓에서 I/O변화가 발생시 select()문이 리턴되어 원하는 작업을 함
 - fcntl()를 사용하여 소켓을 signal-driven I/O모드로 바꾸는 방법
 - 특정 소켓에서 I/O변화가 발생시 SIGIO 시그널을 발생시킴
 - 응용 프로그램에서는 이 시그널을 받아 필요한 작업을 함
- 통신상대가 여러이거나 여러 작업을 병행하여 실행할 경우 사용

4.2 다중처리 기술

다중 입출력 처리를 포함해 네트워크 프로그램에서 네트워크 프로그램에서 동시에 여러 작업을 처리하는 기술

4.2.1 멀티태스킹

멀티태스킹

여러 작업을 병행하여 처리하는 기법
멀티프로세스, 멀티스레드

멀티태스킹

■ 멀티프로세스

- 유닉스에서 멀티태스킹을 위해 프로세스를 여러 개 실행시키는 방법
- 독립적으로 처리해야 할 작업의 수만큼 프로세스를 만드는 방법
- 각 프로세스들이 독립적으로 작업을 처리하므로 구현이 간편함
- 다중처리할 작업이 상당히 독립적으로 진행되어야 할 경우에 적합
- 병렬처리 해야 할 작업 수만큼 프로세스를 생성하는 단점
 - 프로세스가 증가하면 메모리 사용량이 증가하고 프로세스 스케줄링 횟수가 많아짐
 - 동시에 개설할 수 있는 프로세스의 수를 제한함
- 프로세스간 데이터를 공유하기가 불편함
 - 운영체제의 도움을 받아 프로세스간 통신(IPC)를 하므로 프로그램 구현이 복잡해짐

■ 멀티스레드

- 유닉스에서 멀티태스킹을 위해 스레드를 여러 개 실행시키는 방법
- 프로세스 내에서 독립적으로 실행하는 스레드를 여러 개 실행시킴
 - 외부에서는 이 스레드들 전체가 하나의 프로세스처럼 취급됨
- 프로세스에서 스레드를 생성하면 새로 생성된 스레드는 원래 프로세스의 이미지를 같이 사용
- 새로 생성된 스레드용 스택 영역은 스레드별로 별도로 배정되며 스택을 공유하지 않음
- 멀티프로세스 방식보다 필요로하는 메모리양이 적고 스레드 생성 시간이 짧음
- 스레드간 스케줄링도 프로세스간 스케줄링보다 빠르게 이루어짐
- 다중 처리할 작업들이 서로 밀접한 관계로 데이터 공유가 많이 필요한 경우에 적합
- 동기화문제 발생
 - 한 프로세스 내에서 생성된 스레드들이 이미지를 공유하므로 전역 변수를 같이 사용함
 - 스레드들은 쉽게 데이터를 공유할 수 있음
 - 한 스레드가 어떤 변수의 값을 변경하는 도중에 다른 스레드가 동시에 이 변수를 액세스
 - 변수의 값이 명확하게 사용되지 않음

4.2.2 다중화

한 프로세스 또는 스레드 내에서 이루어지는 다중처리 방법

폴링

처리해야 할 작업들을 순차적으로 돌아가면서 처리하는 방법

인터럽트

프로세스가 어떤 작업을 처리하는 도중에 특정한 이벤트가 발생하면 해당 이벤트 처리하는 방법

셀렉팅

폴링의 반대 개념

폴링

- 서버가 여러 클라이언트와 통신할 때 각 클라이언트로부터의 데이터 수신을 순차적으로 처리
- 입출력 함수가 어느 한 곳에서 블록되지 않아야 하므로 파일이나 소켓을 년블록 모드로 설정
- 년블록 모드의 파일이나 소켓
 - 블록될 수 있었던 입출력 함수 호출시 시스템이 즉시 처리하면 결과를 바로 리턴
 - 즉시 처리 할 수 없는 경우엔 함수가 즉시 리턴되어 프로그램이 블록되지 않는 모드
- 여러 클라이언트들이 고르게 트래픽을 발생시키는 경우에 서버에서 사용하기에 적합
 - 서버는 폴링을 할 때마다 수신할 데이터가 거의 항상 기다리고 있기 때문에 프로그램이 효율적으로 실행됨

인터럽트

- 하드웨어 인터럽트
 - 키보드 등을 사용하여 입력하였을 때 발생하는 인터럽트
- 소프트웨어 인터럽트
 - 프로세스 사이에 이벤트 발생을 알려주기 위한 시그널
- 인터럽트 처리
 - CPU에서 하드웨어 인터럽트를 인터럽트 처리 루틴에서 처리함
 - 시그널을 사용하여 데이터 입출력을 인터럽트 방식으로 처리함

선택팅

- 어떤 클라이언트로부터 데이터가 도착하면 서버는 데이터가 도착한 클라이언트와의 입출력 처리
- 클라이언트로부터의 데이터 도착이 불규칙적인 경우에 적합
- 유닉스에서 선택팅을 사용하기 위해 `select()` 함수를 이용
 - 입출력을 처리할 파일이나 소켓들을 비동기 모드로 바꿈

4.3 비동기형 채팅 프로그램

4.3.1 채팅 서버 프로그램 구조

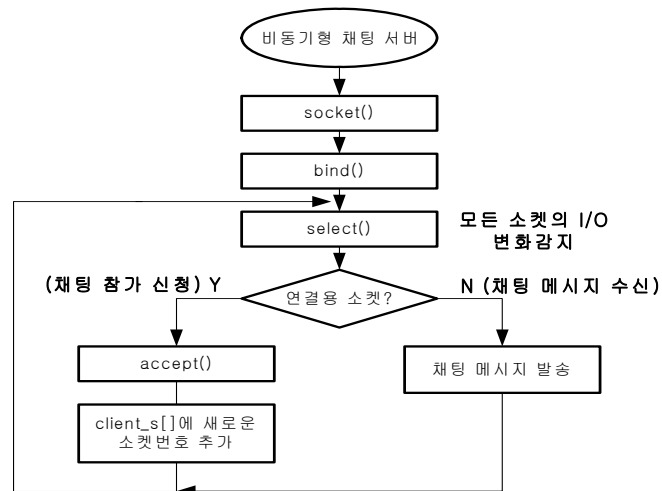


그림 4.1 select()를 이용한 비동기형 채팅 서버

채팅 서버 프로그램 구조

■ 연결용 소켓

- 서버에서 socket()을 호출하여 채팅 참가를 접수할 소켓을 개설
- 이 소켓을 자신의 소켓주소와 bind()함

■ 통신용 소켓

- 연결용 소켓을 대상으로 select()를 호출하여 새로운 참가 요청을 처리
- accept()가 리턴하는 소켓번호를 채팅참가자 리스트에 등록
- 서버는 연결용 소켓과 통신용 소켓을 대상으로 다시 select() 호출

채팅 서버와 클라이언트의 연결관계

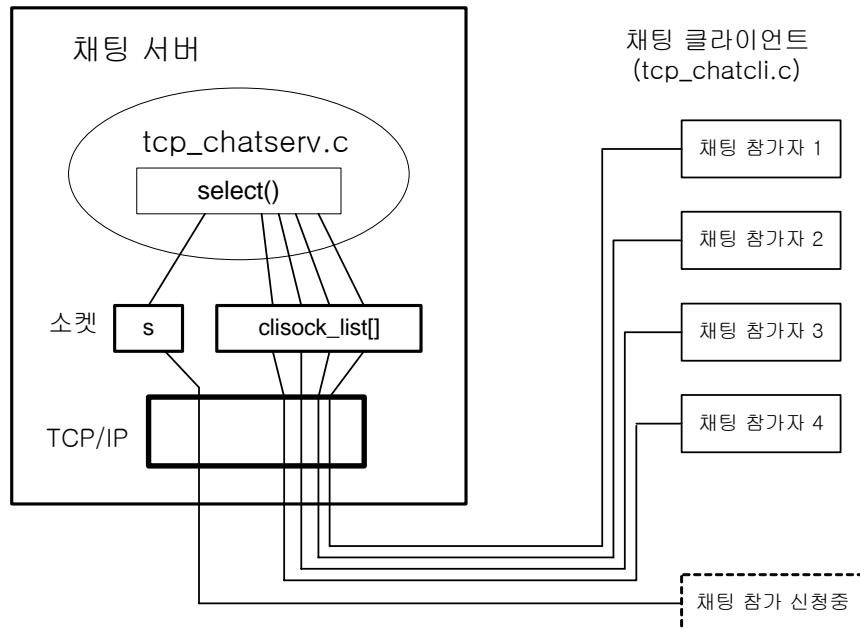


그림 4.2 채팅 서버와 클라이언트의 연결관계

채팅 서버와 클라이언트의 연결관계

- select()
 - 소켓에서 발생하는 I/O변화를 기다리다가 지정된 I/O변화 발생시 리턴
 - 응용 프로그램에서는 select() 리턴시 어떤 소켓에서 어떤 I/O변화인지 확인하고 작업 처리
- 서버
 - 연결용 소켓과 채팅 참가자 리스트를 대상으로 select()를 호출하고 대기
 - select()문이 리턴시
 - 새로운 참가자의 연결요청인지 기존 채팅 참가자중 누군가 채팅 메시지를보낸 것인지 구분하여 필요한 작업 수행

4.3.2 select()

select()의 사용문법

```
int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *tvptr);
```

select()

■ 사용문법

- maxfdp1 : 현재 개설된 소켓번호 중 가장 큰 소켓 번호 +1의 값
- fd_set 타입의 인자 : 각각 읽기, 쓰기, 예외발생 같은 I/O변화시 이를 감지할 대상이 되는 소켓들을 지정하는 배열형 구조체
 - 이 세 구조체를 통해 어떤 소켓에서 어떤 종류의 I/O변화가 발생하는지 감지할지를 선택
- tvptr : select()가 기다리는 시간을 지정
 - NULL인 경우 지정한 I/O변화가 발생할 때까지 무한히 기다림
 - 0인 경우 기다리지 않고 바로 리턴
 - 그 외엔 지정된 시간만큼 또는 도중에 I/O변화가 발생할 때까지 기다림

fd_set 타입의 구조체와 소켓번호와의 관계

소켓번호 :	0	1	2	3		maxfdp1-1	
readfds :	1	0	0	1	...	0	1비트 어레이
writefds :	0	1	0	1	...	0	
exceptfds :	0	0	0	0	...	0	

그림 4.3 fd_set 타입의 구조체와 소켓번호와의 관계

fd_set 타입의 구조체와 소켓번호와의 관계

■ 동작방식

- fd_set 타입 구조체에 I/O변화를 감지할 소켓이나 파일을 1로 set
- select()를 호출해 두면 해당 조건이 만족되는 순간 select()문이 리턴

■ readfds

- 0,3번이 세트되어 있음
- 키보드(표준입력 0), 소켓번호 3에서 어떤 데이터가 입력시
→ 프로그램이 이를 읽을 수 있는 상태가 되면 select()문이 리턴

■ writefds

- 1, 3번이 세트되어 있음
- 파일기술자(표준출력 1)나 소켓번호 3번이 write를 할 수 있는 상태로 변할시
- select()문이 리턴

fd_set을 사용하기 위한 매크로

문법
FD_ZERO(fd_set *fdset)
FD_SET(int fd, fd_set *fdset)
FD_CLR(int fd, fd_set *fdset)
FD_ISSET(int fd, fd_set *fdset)

표 4.1 fd_set을 사용하기 위한 매크로

fd_set을 사용하기 위한 매크로

- fd_set 타입 구조체의 배열 값을 편리하게 지정하기 위해 매크로가 제공됨
- FD_ZERO(fd_set *fdset)
 - fdset의 모든 비트를 지움
- FD_SET(int fd, fd_set *fdset)
 - fdset 중 소켓 fd에 해당하는 비트를 1로 함
- FD_CLR(int fd, fd_set *fdset)
 - fdset 중 소켓 fd에 해당하는 비트를 0으로 함
- FD_ISSET(int fd, fd_set *fdset)
 - fdset 중 소켓 fd에 해당하는 비트가 set되어 있으면 양수 값을 리턴

4.3.3 채팅 서버 프로그램

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <sys/socket.h>
6 #include <sys/file.h>
7 #include <netinet/in.h>
8 #include <string.h>
9
10 #define MAXLINE 512
11 #define MAX SOCK 64
12
13 char *escapechar = "exit";
14 int getMax(int);
15 void exitClient(int);
16 int maxfdp1;
17 int num_user = 0;
18 int sock_list[MAX SOCK];
19
```

```

20 int main(int argc, char *argv[]){
21     char rline[MAXLINE], msg[MAXLINE];
22     char *start = "채팅 서버에 접속되었습니다.Wn";
23     int i, j, n;
24     int s, client_fd, client_len;
25
26     fd_set  read_fds;
27     struct sockaddr_in  client_addr, server_addr;
28
29     if(argc != 2){
30         printf("사용법 :%s portWn", argv[0]);
31         exit(0);
32     }
33
34     if((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
35         printf("서버 : 스트림 소켓을 열 수 없습니다. .");
36         exit(0);
37     }
38
39     bzero((char *)&server_addr, sizeof(server_addr));
40     server_addr.sin_family = AF_INET;
41     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
42     server_addr.sin_port = htons(atoi(argv[1]));
43

```

- 26 fd_set read_fds;
 - 읽기를 감지할 소켓번호 구조체
- 34 ~ 37번 줄
 - 초기 소켓 생성
- 39 ~ 42번 줄
 - server_addr 구조체의 내용 세팅

```

44     if (bind(s,(struct sockaddr *)&server_addr,sizeof(server_addr)) < 0) {
45         printf("서버 : local address에 bind 할 수 없습니다.\n");
46         exit(0);
47     }
48
49     listen(s, 5);
50
51     maxfdp1 = s + 1;
52
53     while(1) {
54         FD_ZERO(&read_fds);
55         FD_SET(s, &read_fds);
56         for(i=0; i<num_user; i++) FD_SET(sock_list[i], &read_fds);
57         maxfdp1 = getMax(s) + 1;
58         if (select(maxfdp1, &read_fds, (fd_set *)0, (fd_set *)0,(struct timeval *)0) < 0) {
59             printf("select error <= 0 \n");
60             exit(0);
61         }
62         if(FD_ISSET(s, &read_fds)) {
63             client_len = sizeof(client_addr);
64             client_fd = accept(s, (struct sockaddr *)&client_addr, &client_len);
65
66             if(client_fd == -1) {
67                 printf("accept error\n");
68                 exit(0);
69             }
70

```

- 49 listen(s, 5);
 - 클라이언트로부터 연결 요청을 기다림
- 51 maxfdp1 = s + 1;
 - 최대 소켓번호 + 1
- 57 maxfdp1 = getMax(s) + 1;
 - maxfdp1 재계산

```

71     sock_list[num_user] = client_fd;
72     num_user++;
73     send(client_fd, start, strlen(start), 0);
74     printf("%d번째 사용자 추가.\n", num_user);
75 }
76
77 for(i = 0; i < num_user; i++) {
78     if(FD_ISSET(sock_list[i], &read_fds)) {
79         if((n = recv(sock_list[i], rline, MAXLINE, 0)) <= 0) {
80             exitClient(i);
81             continue;
82         }
83
84         if(strstr(rline, escapechar) != NULL) {
85             exitClient(i);
86             continue;
87         }
88
89         rline[n] = '\0';
90         for (j = 0; j < num_user; j++) send(sock_list[j], rline, n, 0);
91         printf("%s\n", rline);
92     }
93 }
94 }
95 }
96

```

- 71 sock_list[num_user] = client_fd;
 - 채팅 클라이언트 목록에 추가
- 77 ~ 82번 줄
 - 클라이언트가 보낸 메시지를 모든 클라이언트에게 방송
- 84 ~ 87번 줄
 - 종료 문자 처리
- 80 ~ 91번 줄
 - 모든 채팅 참가자에게 메시지 방송

```

97 void exitClient(int i) {
98     close(sock_list[i]);
99     if(i != num_user-1) sock_list[i] = sock_list[num_user-1];
100     num_user--;
101     printf("채팅 참가자 1명 탈퇴하였습니다. 현재 참가자 수는 %d명 입니다.\n", num_user);
102 }
103
104 int getMax(int k) {
105     int max = k;
106     int r;
107     for (r=0; r < num_user; r++) {
108         if (sock_list[r] > max ) max = sock_list[r];
109     }
110     return max;
111 }

```

- 97 ~ 102번 줄
 - 채팅 탈퇴 처리

- 104 ~ 111번 줄
 - clien_s[] 내의 최대 소켓번호 얻기(초기치는 k)

실행화면

```
$ tcp_chatseru 9999
1번째 사용자 추가.
[linux] 테스트1입니다.

2번째 사용자 추가.
채팅 참가자 1명 탈퇴하였습니다. 현재 참가자 수는 1명 입니다.
2번째 사용자 추가.
[test] 테스트2입니다.

채팅 참가자 1명 탈퇴하였습니다. 현재 참가자 수는 1명 입니다.
```

실행

- 파일명 : chat_server.c
- 컴파일 : gcc -o chat_server chat_server.c
- 사용법 : chat_server 포트번호

통신용 소켓 구분

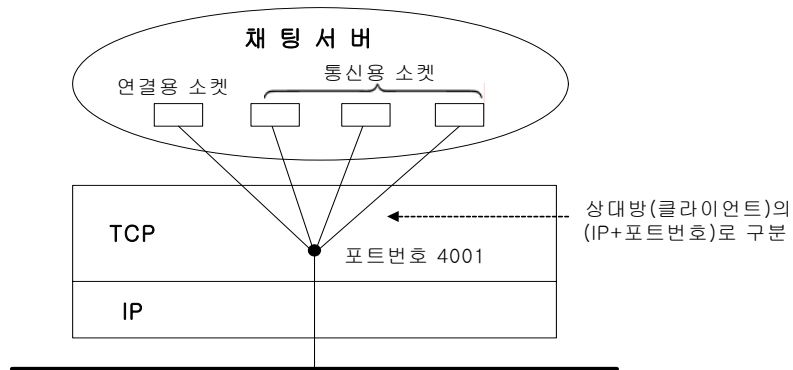


그림 4.4 TCP서버에세의 다수의 통신용 소켓을 구분하는 방법

통신용 소켓 구분

- 채팅 서버는 클라이언트와 통신에 사용하기 위해 다수의 통신용 소켓을 개설
 - 모든 소켓은 같은 포트번호를 사용함
 - 클라이언트들은 서버의 포트번호만 알고 있음
 - 서버에서 다른 포트번호를 사용하면 통신이 안됨
- 한 개의 포트번호만으로 각 클라이언트와 연결할 수 있는 이유
 - 클라이언트의 IP 주소와 포트번호를 내부적인 키로 사용
 - 채팅 서버에 도착하는 데이터그램의 포트번호는 모두 같음
 - 서버는 클라이언트의 IP 주소와 포트번호를 키 값으로 사용하여 해당 통신용 소켓으로 메시지 전달
- 두 개의 telnet 클라이언트가 서버에 접속시
 - 서버에 두 개의 통신용 소켓이 생성됨
 - 통신용 소켓은 모두 23번을 포트번호로 사용함
 - 각 클라이언트를 구분하기 위해 클라이언트의 IP 주소와 포트번호를 키로 사용

4.3.4 채팅 클라이언트 프로그램

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <sys/time.h>
7 #include <string.h>
8
9 #define MAXLINE 512
10 #define MAX_SOCKET 128
11
12 char *escapechar = "exit";
13 char name[10];
14
15 int main(int argc, char *argv[]) {
16     char line[MAXLINE], msg[MAXLINE+1];
17     int n, pid;
18     struct sockaddr_in server_addr;
19     int maxfdp1;
20     int s;
21 }
```

- 13 char name[10];
 - 채팅에서 사용할 이름을 저장
- 20 int s;
 - 서버와 연결된 소켓번호

```

22  fd_set read_fds;
23
24  if(argc != 4) {
25      printf("사용법 : %s sever_IP port name Wn", argv[0]);
26      exit(0);
27  }
28
29  sprintf(name, "[%s]", argv[3]);
30
31  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
32      printf("클라이언트 : 스트림 소켓을 열수 없습니다..Wn");
33      exit(0);
34  }
35
36  bzero((char *)&server_addr, sizeof(server_addr));
37  server_addr.sin_family = AF_INET;
38  server_addr.sin_addr.s_addr = inet_addr(argv[1]);
39  server_addr.sin_port = htons(atoi(argv[2]));
40
41  if(connect(s, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
42      printf("클라이언트 : 서버에 접속할 수 없습니다.Wn");
43      exit(0);
44  } else {
45      printf("서버에 접속되었습니다. Wn");
46  }
47

```

- 29 `sprintf(name, "[%s]", argv[3]);`
 - 채팅 참가자 이름 구조체 초기화
- 31 ~ 34번 줄
 - 소켓 생성
- 36 ~ 39번 줄
 - 채팅 서버의 소켓주소 구조체 `server_addr` 초기화
- 41 ~ 46번 줄
 - 연결 요청

```

48     maxfdp1 = s + 1;
49     FD_ZERO(&read_fds);
50
51     while(1) {
52         FD_SET(0, &read_fds);
53         FD_SET(s, &read_fds);
54
55         if(select(maxfdp1, &read_fds, (fd_set *)0, (fd_set *)0, (struct timeval *)0) < 0) {
56             printf("select error\n");
57             exit(0);
58         }
59
60         if (FD_ISSET(s, &read_fds)) {
61             int size;
62             if ((size = recv(s, msg, MAXLINE, 0)) > 0) {
63                 msg[size] = '\0';
64                 printf("%s\n", msg);
65             }
66         }
67
68         if (FD_ISSET(0, &read_fds)) {
69             if(fgets(msg, MAXLINE, stdin)) {
70                 sprintf(line, "%s %s", name, msg);
71
72                 if (send(s, line, strlen(line), 0) < 0)
73                     printf("Error : Write error on socket.\n");
74
75                 if (strstr(msg, escapechar) != NULL ) {
76                     printf("종료되었습니다.\n");
77                     close(s);
78                     exit(0);
79                 }
80             }
81         }
82     }
83 }

```

실행화면

참가자1

```
$ tcp_chatcli 210.115.49.220 9999 linux
서버에 접속되었습니다.
채팅 서버에 접속되었습니다.

테스트1입니다.
[linux] 테스트1입니다.

[test] 테스트2입니다.
```

참가자2

```
$ tcp_chatcli 210.115.49.220 9999 test
서버에 접속되었습니다.
채팅 서버에 접속되었습니다.

테스트2입니다.
[test] 테스트2입니다.
```

실행

- 파일명 : chat_client.c
- 컴파일 : gcc -o chat_client chat_client.c
- 사용법 : chat_client 서버IP 포트번호 접속할 이름

4.4 폴링형 채팅 프로그램

소켓을 년블록모드로 설정하고 채팅 메시지 수신여부를 폴링방식으로 점검

4.4.1 fcntl()

소켓을 년블록 모드나 비동기 모드로 바꿈

fcntl()

- file control을 의미함
- fcntl()의 동작
 - 소켓을 년블록 모드로 설정
 - 소켓을 비동기 모드로 설정
 - 소켓의 소유자를 설정하거나 현재 소유자를 얻어옴

넌블록 모드 설정

문법

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, long flag);
```

설정 코드

```
int val

if ((val = fcntl(sock_fd, F_GETFL, 0)) < 0)
    exit(1);

val |= O_NONBLOCK;

if ((fcntl, F_SETFL, val) < 0)
    exit(1);
```

넌블록 모드 설정

■ 문법

- fd : 모드 변경을 원하는 소켓 디스크립터
- cmd(명령)
 - F_SETFL : 플래그 세트
 - F_GETFL : 플래그 읽기
 - F_SETOWN : 소켓의 소유자 설정
 - F_GETOWN : 소켓의 소유자 얻기
- flag
 - 구체적인 옵션을 설정
 - O_NONBLOCK : 넌블록 모드로 설정
 - O_ASYNC : SIGIO 시그널에 의해 구동되도록 비동기 모드로 설정

■ 설정 코드

- 소켓을 넌블록 모드로 설정하는 코드
- 기존의 플래그 값을 유지
 - F_GETFL 명령으로 얻은 후 이를 O_NONBLOCK 와 OR 연산

비동기 모드 설정

소켓을 비동기 모드로 바꾸는 코드

```
int flag;

if ((flag = fcntl(fd, F_GETFL, 0)) < 0)
    exit(1);

Flag |= O_ASYNC;

If ((fcntl, F_SETFL, flag) < 0)
    exit(1);
```

소켓의 소유자를 설정하거나 현재 소유자를 얻어옴

코드

```
fcntl(fd, F_SETOWN, getpid());
Fcntl(fd, F_GETOWN, &pid);
```

비동기 모드 설정

- 비동기 모드로 설정하는 이유
 - select() 함수 자체는 블록형 함수임
 - 아무 입출력 변화가 없으면 프로그램은 select()에서 블록
 - 인터럽트형 다중화에서는 SIGIO 등의 시그널이 발생할 때 입출력을 처리할 수 있음

소켓의 소유자를 설정하거나 현재 소유자를 얻어옴

- 소켓의 소유자를 설정하는 이유
 - 소켓을 처음 생성하면 소유자가 없음
 - 프로세스에서 나중에 시그널을 수신하도록 하려면 소켓의 소유자가 필요
 - 소켓의 소유자에게 SIGIO나 SIGURG 등의 시그널이 전달됨
- accept()가 리턴하는 통신용 소켓은 연결용 소켓의 소유자를 상속받음

4.4.2 폴링형 채팅 서버

넢블록 모드

폴링형 채팅 서버 프로그램에서는 소켓들을 넢블록 모드로 설정한 후 무한 루프로 입출력 폴링

넢블록 모드

- 소켓에 대해 `read()`, `write()`등의 입출력 함수를 호출하면 함수는 바로 리턴
 - 함수의 리턴 값을 보고 원하는 작업의 실행여부를 확인
 - 정상인 경우 0, 에러인 경우 -1이 리턴
 - 에러인 경우 에러코드 값을 보고 함수 자체의 에러인지 소켓이 즉시 리턴된 것인지 확인
 - 소켓이 즉시 리턴된 것은 소켓이 넢블록 모드이므로 리턴됨
 - 이 경우 에러코드는 `EWOULDBLOCK`
- `recv()`를 호출했을 때 `EWOULDBLOCK` 이외의 에러
 - 클라이언트와의 연결 종료 또는 클라이언트가 리셋을 보낸 경우
 - 클라이언트를 채팅 목록에서 제거하는 등의 에러처리를 함
- `accept()`
 - 넢블록 모드의 소켓에 `accept()`를 호출한 경우
 - `accept()`가 리턴한 소켓은 디폴트로 블록 모드임
 - 필요한 경우 명시적으로 넢블록 모드로 변환해야 함

소켓의 모드 확인

사용자 정의 함수 is_nonblock()

```
int is_nonblock(int sockfd)
{
    int val

    val = fcntl(sockfd, F_GETFL, 0);

    if(val & O_NONBLOCK)
        return 0;
    return -1;
}
```

소켓의 모드 확인

- val = fcntl(sockfd, F_GETFL, 0);
 - 기존의 플래그 값을 얻어옴

- if(val & O_NONBLOCK)
 - 논블록 모드인지 확인
 - 논블로 모드일 경우 0을 리턴, 아니면 -1을 리턴

폴링형 채팅 서버 프로그램

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <fcntl.h>
5 #include <signal.h>
6 #include <arpa/inet.h>
7 #include <sys/types.h>
8 #include <sys/file.h>
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <unistd.h>
12 #include <errno.h>
13
14 #define MAXLINE 512
15 #define MAX_SOCKET 128
16
17 char *escapechar = "exit";
18 char *join_MSG = "서버에 접속하였습니다.\n";
19
20 int maxfdp1;
21 int num_user = 0;
22 int clisock_list[MAX_SOCKET];
23 int server_sock;
24
```

- 20 int maxfdp1;
 - 최대 소켓번호 +1
- 21 int num_user = 0;
 - 채팅 참가자수 초기화
- 22 int clisock_list[MAX_SOCKET];
 - 채팅 참가자 소켓번호 목록

```

25 void joinClient(int s, struct sockaddr_in *newclient_addr);
26 void exitClient(int);
27 int set_nonblock(int sockfd);
28 int is_nonblock(int sockfd);
29 int sock_listen(int host, int port, int backlog);
30
31 void errquit(char *msg){
32     perror(msg);
33     exit(1);
34 }
35 int main(int argc, char *argv[])
36 {
37     char buf[MAXLINE];
38     int i , j, nbyte;
39     int accp_sock, client_len;
40     struct sockaddr_in client_addr;
41
42     if(argc != 2){
43         printf("사용법 : %s portWn", argv[0]);
44         exit(0);
45     }
46

```

```

47  server_sock = sock_listen(INADDR_ANY, atoi(argv[1]),5);
48  if(server_sock == -1)
49      errquit("소켓 리슨 실패");
50  if(set_nonblock(server_sock) == -1)
51      errquit("넌블록 셋팅 실패");
52
53  while(1){
54      client_len = sizeof(client_addr);
55      accp_sock = accept(server_sock,(struct sockaddr *)&client_addr,&client_len);
56      if(accp_sock == -1 && errno != EWOULDBLOCK)
57          errquit("accept fail");
58      else if(accp_sock >0){
59          clisock_list[num_user] = accp_sock;
60
61          if(is_nonblock(accp_sock) != 0 &&set_nonblock(accp_sock)<0)
62              errquit("넌블록 셋팅 실패");
63
64          joinClient(accp_sock,&client_addr);
65          send(accp_sock, join_MSG, strlen(join_MSG),0);
66          printf("%d번째 사용자 추가.\n", num_user);
67      }
68
69      for(i= 0; i<num_user ; i++)
70      {
71          errno = 0;
72          nbyte = recv(clisock_list[i], buf, MAXLINE, 0);
73          if(nbyte == 0){

```

- 59 clisock_list[num_user] = accp_sock;
 - 채팅 클라이언트 목록에 추가

- 61 ~ 62번 줄
 - 통신용 소켓은 넌블록 모드가 아님

- 69 ~ 90번 줄
 - 클라이언트가 보낸 메시지를 모든 클라이언트에게 방송

```

74         exitClient(i);
75         continue;
76     }
77     else if(nbyte == -1 && errno == EWOULDBLOCK)
78         continue;
79
80     if(strstr(buf, escapechar) != NULL){
81         exitClient(i);
82         continue;
83     }
84
85     buf[nbyte] = 0;
86
87     for(j=0;j<num_user;j++)
88         send(clisock_list[j], buf, nbyte, 0);
89     printf("%s\n", buf);
90 }
91 }
92 }
93
94 void joinClient(int s,struct sockaddr_in *newclient_addr){
95     char buf[20];
96     inet_ntop(AF_INET,&newclient_addr->sin_addr, buf,sizeof(buf));
97     printf("new client : %s\n", buf);
98
99     clisock_list[num_user] = s;
100     num_user++;
101 }

```

- 74 exitClient(i);
 - abrupt exit
- 81 exitClient(i);
 - abrupt exit
- 85 ~ 89번 줄
 - 모든 채팅 참가자에게 메시지 발송
- 94 ~ 101번 줄
 - 새로운 채팅 참가자 처리
- 99 clisock_list[num_user] = s;
 - 채팅 클라이언트 목록에 추가

```

103 void exitClient(int i){
104     close(clisock_list[i]);
105     if(i != num_user-1)
106         clisock_list[i]=clisock_list[num_user-1];
107     num_user--;
108     printf("채팅 참가자 1명 탈퇴하였습니다. 현재 참가자 수 %d명 입니다.\n", num_user);
109 }
110
111 int is_nonblock(int sockfd)
112 {
113     int val;
114
115     val = fcntl(sockfd, F_GETFL,0);
116
117     if(val & O_NONBLOCK)
118         return 0;
119     return -1;
120 }
121
122 int set_nonblock(int sockfd)
123 {
124     int val;
125
126     val=fcntl(sockfd, F_GETFL,0);
127
128     if(fcntl(sockfd, F_SETFL, val | O_NONBLOCK) == -1)
129         return -1;
130     return 0;
131 }

```

- 103 ~ 109번 줄
 - 채팅 탈퇴 처리
- 111 ~ 120번 줄
 - 소켓이 nonblock 인지 확인
- 115 val = fcntl(sockfd, F_GETFL,0);
 - 기존의 플래그 값을 얻어옴
- 117 ~ 118번 줄
 - 년블록 모드인지 확인
- 122 ~ 131번 줄
 - 소켓을 년블록 모드로 설정
- 126 val=fcntl(sockfd, F_GETFL,0);
 - 기존의 플래그 값을 얻어옴

```

133 int sock_listen(int host, int port, int backlog)
134 {
135     int sd;
136     struct sockaddr_in server_addr;
137
138     sd = socket(AF_INET, SOCK_STREAM, 0);
139     if(sd == -1)
140     {
141         perror("socket fail");
142         exit(1);
143     }
144
145     bzero((char *)&server_addr, sizeof(server_addr));
146     server_addr.sin_family = AF_INET;
147     server_addr.sin_addr.s_addr = htonl(host);
148     server_addr.sin_port = htons(port);
149
150     if(bind(sd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0){
151         perror("bind fail");
152         exit(1);
153     }
154
155     listen(sd, backlog);
156     return sd;
157 }

```

- 133 ~ 157번 줄
 - listen 소켓 생성 및 listen
- 145 ~ 148번 줄
 - servaddr 구조체의 내용 세팅
- 165 listen(sd, backlog);
 - 클라이언트로부터 연결요청을 기다림

실행화면

서버 실행

```
$ non_chaterv 9999
new client : 210.115.49.220
1번째 사용자 추가.
[linux] 테스트입니다.

new client : 210.115.49.220
2번째 사용자 추가.
[test] 테스트입니다.

채팅 참가자 1명 탈퇴하였습니다. 현재 참가자 수 1명 입니다.
```

참가자1

```
$ tcp_chatcli 210.115.49.220 9999 linux
서버에 접속되었습니다.
서버에 접속하였습니다.

테스트입니다.
[linux] 테스트입니다.

[test] 테스트입니다.
```

참가자2

```
$ tcp_chatcli 210.115.49.220 9999 test
서버에 접속되었습니다.
서버에 접속하였습니다.

테스트입니다.
[test] 테스트입니다.

exit
Good bye.
```

실행

- 파일명 : nonb_chatserv.c
- 컴파일 : gcc -o nonb_chatserv nonb_chatserv.c
- 사용법 : nonb_chatserv 포트번호
- 클라이언트 프로그램은 앞의 chat_client.c를 사용
 - chat_client IP주소 포트번호 ID

5장

소켓 옵션

5.1 소켓 옵션 종류

5.2 소켓 옵션 변경

Overview

- TCP/IP 프로토콜의 세부적인 기능 활용을 위한 방법 습득
- 소켓의 동작 특성을 변경

5.1 소켓 옵션 종류

5.1.1 SO_KEEPALIVE

TCP 연결이 정상적으로 지속되고 있는지 주기적으로 확인

사용방법

```
int set = 1;
setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, (char*)&set, sizeof(on));
```

SO_KEEPALIVE

■ 동작방식

- 옵션이 셋팅되어 있으면 TCP는 확인시간(예: 2시간)동안 데이터 송수신을 기다림
→ 없을시 TCP연결이 아직 살아있는지 질문(keep alive prove)을 보냄
- Keep alive 질문에 대해 중간 라우터가 ICMP 메시지로 host unreachable error 또는 network unreachable error을 보낸 경우
→ EHOSTENREACH 또는 ENETUNREACH 에러가 리턴

■ keep alive prove의 응답

- 상대방이 ACK를 보냄
→ TCP 연결이 정상적으로 동작 중이므로 확인 시간 이후에 질문을 다시 보냄
- 상대방이 RST 에러를 보냄
→ 상대방 호스트가 꺼진 후 재부팅 된 상태
→ TCP는 소켓을 닫으며 ECONNRESET의 에러코드를 가짐
- 아무 응답이 없음
→ 질문을 몇 번 더 보낸 후 연결을 종료
→ 에러코드는 ETIMEDOUT

■ 사용 용도

- 비정상적인 TCP연결 종료를 찾아 소켓을 종료하기 위해 사용

5.1.2 SO_LINGER

close()를 호출 후에 상대방에서도 정상적으로 종료 절차가 이루어지는지 확인

사용방법

```
struct linger{
    int l_onoff;
    int l_linger;
};

struct linger ling;
ling.l_onoff = 1;
ling.l_linger = 0;
if(setsockopt(sockd, SOL_SOCKET, SO_LINGER, (void*) &ling, sizeof(struct linger)) != 0){
    perror("소켓 옵션 지정 실패");
    exit(1);
}
```

SO_LINGER

■ 동작방식

- close()를 호출한 이후에 상대방에서도 정상적으로 종료 절차가 이루어 졌는지 사용
- TCP연결을 종료할 때 close()를 호출하여 종료함
 - close()는 즉시 리턴됨
 - 소켓 송신버퍼에 전송할 데이터가 있어도 close()는 리턴되며 모두 전송 후 연결 종료
 - close()가 리턴되어도 모든 데이터의 전송 여부는 보장되지 않음
- close()는 옵션에서 지정한 linger 시간 또는 정상 종료 때까지 블록
 - 정상 종료 전에 linger 시간이 먼저 타임 아웃되면 ETIMEDOUT 에러 발생
- 송신버퍼에 남아있던 데이터를 전송하기까지 기다리는 최대 시간을 제한하는데 사용
 - close()는 linger 시간 이내에 데이터를 전송하지 못하면 에러를 리턴하고 통신이 종료 되기 때문

■ 사용방법

- 송신버퍼에 데이터가 남아 있어도 타임아웃이 되면 데이터를 폐기하도록 옵션을 지정
- `l_onoff`는 LINGER 옵션의 사용 여부를 지정
 - `l_onoff = 0` 일 때 LINGER 옵션을 사용하지 않음
 - `close()`는 원래대로 동작하여 송신버퍼에 전송할 데이터가 남아있어도 즉시 리턴
 - 송신버퍼에 데이터가 남아 있었다면 커널에 의해 전송이 마무리됨
 - 전송이 실패시 프로그램에서 확인할 수 없음
 - 연결 종료후 TCP가 2MSL 시간동안 TIME-WAIT 상태가 됨
- `l_linger`는 기다리는 시간을 지정
 - `l_linger = 0` 일 때 송신버퍼에 남아 있는 데이터는 파기
 - 정상적인 종료 과정을 거치지 않고 RST을 전송하여 TCP 연결을 즉시 리셋
 - 이 때는 TCP가 TIME-WAIT 상태에도 머무르지 않음
 - `l_linger`이 양수 값일 때, 송신버퍼에 데이터가 있는 경우 `close()`는 데이터를 모두 전송하고 ACK를 수신할 때까지 또는 `l_linger` 시간이 지날 때까지 블록
 - 소켓이 nonblock 모드일 경우 `close()` 함수는 즉시 리턴
 - `l_linger` 시간이 먼저 지나서 `close()`가 리턴된 경우에는 송신버퍼에 남은 데이터는 폐기

shutdown()

close()대신 상대방이 FIN을 보낼 때 까지 기다리는 방법

사용방법

```
int shutdown(int s, int how);
```

shutdown()

■ 동작방식

- 호스트 A가 호스트 B로 데이터를 보낸 후 close()를 호출한 시점에 아직 호스트 B의 수신버퍼에 호스트 A가 보낸 데이터가 읽히지 않고 대기하고 있는 경우
 - 호스트 B는 수신버퍼에 아직 읽지 않은 데이터가 있어도 호스트 A가 보내온 FIN에 대한 ACK를 보냄
 - 호스트 A의 close()함수가 정상적으로 리턴되어도 호스트 B의 응용 프로그램이 호스트 A가 보낸 모든 데이터를 읽었는지는 모름
 - 이 경우 shutdown()을 사용하여 상대방이 FIN을 보낼 때 까지 대기

■ 사용방법

- int s
 - 소켓번호
- int how
 - 양방향 스트림에 대한 동작을 지정하는 인자
 - SHUT_WR : 송신 스트림만 닫음
 - SHUT_RD : 수신 스트림만 닫음
 - SHUT_RDWR : 송신 및 수신 양방향 스트림을 닫음

5.1.3 SO_RCVBUF와 SO_SNDBUF

소켓의 송신버퍼와 수신버퍼의 크기를 변경할 때 사용

SO_RCVBUF와 SO_SNDBUF

■ 동작방식

- 송신버퍼

- TCP의 경우 응용 프로그램에서 write() 함수를 호출시 커널은 데이터를 소켓 송신버퍼로 복사
- 데이터가 송신버퍼에 모두 복사되면 데이터를 전송하기 시작
- 전송한 후에 바로 송신버퍼를 지우지 않고 재전송의 가능성을 위해 저장하고 있다가 ACK를 받은 후에 송신버퍼를 지움
- 송신버퍼가 가득 차게 되면 write()함수는 블록됨

- 수신버퍼

- TCP는 흐름제어를 위해 수신버퍼의 여유 공간을 상대방에게 알려줌
- 여유 공간의 크기를 윈도우라고 함
- TCP에서는 수신버퍼의 용량보다 큰 데이터를 받게 되면 나머지는 버려짐
- UDP에서는 윈도우를 이용한 흐름제어가 없어 큰 데이터 전송시 주의해야함
- UDP의 수신버퍼의 크기는 보통 40000바이트 정도이며 최소한 576 바이트 이상은 보장

■ 버퍼의 크기 변경 시기

- 클라이언트 : connect() 함수 호출 이전
- 서버 : listen() 함수 호출 이전

5.1.4 SO_REUSEADDR

동일한 소켓주소의 중복 사용을 허용

사용방법

```
int set = 1;
setsockopt(udp_sock1, SOL_SOCKET, SO_REUSEADDR, (void*)&set, sizeof(int));
```

TIME-WAIT 상태에서의 주소 재사용

재사용 옵션을 설정해 사용

자식 프로세스가 서버인 경우

포트번호가 재사용될 수 있도록 옵션을 지정

멀티플 서버의 경우

소켓주소 재사용 옵션 설정해 사용

완전 중복 바인딩

완전 중복 바인딩 된 소켓주소를 사용

SO_REUSEADDR

■ 동작방식

- 동일한 소켓주소를 여러 프로세스나 한 프로세스내의 여러 소켓에서 중복 사용을 허용
- 기본적으로는 한 호스트 내에서 같은 포트번호를 중복하여 사용할 수 없음
- 주로 서버에서 사용됨

■ TIME-WAIT 상태에서의 주소 재사용

- active close를 하면 TCP가 TIME-WAIT 상태에서 2MSL 시간 동안 기다림
→ 이 동안 사용 중이던 포트번호를 중복하여 사용할 수 없음
- bind()를 호출하기 전에 SO_REUSEADDR 옵션을 지정하여 TIME-WAIT 상태에서 포트번호를 중복하여 사용함
- 단, TIME-WAIT 상태에 있을 때만 주소 재사용이 가능함
→ 소켓이 close되지 않은 상태에서는 소켓주소를 중복하여 사용할 수 없음
- TCP가 TIME-WAIT 상태 동안에도 서비스를 즉시 재시작하기 위해 사용

■ 자식 프로세스가 서버인 경우

- TCP 서버 프로그램에서 자식 프로세스가 서비스 처리를 담당하던 중 부모 프로세스가 재시작시
→ 포트번호 사용중 에러가 발생
→ 과거의 자식 프로세스가 해당 포트번호를 사용하기 때문에 발생
- 소켓 개설후 bind() 호출전 옵션 지정

■ 멀티홈 서버의 경우

- 멀티홈 호스트
→ 호스트가 두 개 이상의 랜에 접속된 경우에 호스트가 두 개 이상의 IP 주소를 가짐
- 멀티홈 호스트에서 두 개 이상의 IP 주소가 같은 포트번호를 사용하는 것이 필요한 경우
→ 소켓주소 재사용 옵션 설정
→ 같은 포트번호를 사용하면서 둘 이상의 다른 IP 주소를 사용하는 서버 프로그램들을 한 호스트 내에서 실행시킬 때 소켓주소 재사용 옵션 설정이 필요
- 하나의 프로세스 내에서 여러 소켓을 개설하고 각 소켓들이 같은 포트번호 다른 IP 주소를 사용할 때도 옵션 설정이 필요
→ 각 소켓별로 각각 다른 IP 주소를 지정하여 bind() 함

■ 완전 중복 바인딩

- 여러 프로세스에서 동일한 IP 주소와 동일한 포트번호를 중복하여 bind()하는 것
→ UDP 소켓에서만 사용 가능함
- 사용 예
→ 멀티 캐스트 데이터그램을 수신하는 경우
- 하나의 호스트 내에서 여러 멀티캐스트 가입자들이 데이터를 수신하려고 할 때
→ 각 가입자들이 동일한 데이터그램을 수신하기 위해 완전 중복 바인딩 된 소켓주소를 사용

5.1.5 기타 옵션

SO_OOBINLINE

상대방이 보낸 대역외(OOB : Out Of Band) 데이터를 처리하는 옵션

SO_RCVLOWAT와 SO_SNDLOWAT

송수신버퍼의 최소량 설정하는 옵션

SO_OOBINLINE

- 대역외 데이터를 일반 데이터의 수신버퍼에 같이 저장되도록 함
 - 동등한 순서로 처리됨
 - 일반적으로 대역외 데이터는 일반 데이터보다 우선순위가 높은 버퍼에 저장됨

SO_RCVLOWAT와 SO_SNDLOWAT

- 수신버퍼 최소량은 기본적으로 TCP와 UDP에서 한 바이트임
 - select() 함수가 데이터 읽기 조건이 만족되어 리턴하기 위해 한 바이트 이상의 데이터가 수신버퍼에 도착해야함
 - 한 바이트 이상의 데이터가 수신되면 select() 함수가 리턴됨
- 송신버퍼의 최소량은 기본적으로 2048 바이트임

SO_DONTROUTE

데이터그램 선택시 시스템이 배정하는 라우팅 경로를 사용하지 않도록 하는 옵션

SO_BROADCAST

소켓으로 브로드캐스트 메시지를 보낼 수 있는지를 지정

SO_DONTROUTE

- send(), sendto(), sendmsg() 호출 시에 사용
- 라우팅 테이블이 잘못 되었을 때, 라우팅 테이블을 무시할 때 사용
 - 데이터그램을 로컬 인터페이스에 있는 장비로 전송

SO_BROADCAST

- 브로드캐스트는 UDP 소켓에서만 사용 가능함
- 이더넷 같은 방송형 서브 네트워크에서만 가능함

TCP_NODELAY

Nagle's 알고리즘을 실행시키지 않음을 지정

TCP_MAXSEG

TCP의 최대 세그먼트 크기를 읽거나 변경할 때 사용

TCP_NODELAY

- Nagle's 알고리즘이 디폴트로 실행되므로 이를 취소해야 할 때 사용
- Nagle's 알고리즘
 - 앞에 전송된 데이터가 ACK를 받기 전까지는 작은 크기의 데이터는 전송하지 않고 기다림
 - ACK를 기다리는 동안 작은 데이터들을 모아서 전송
 - 너무 작은 데이터 세그먼트들이 자주 발생하지 않게 하여 전송 효율 높임
 - ACK가 빨리 오면 작은 데이터 송신도 이에 비례하므로 통신 처리가 늦어지지 않음
- delayed ACK 알고리즘
 - 데이터를 수신한 경우 그 데이터에 대해 ACK를 즉시 보내지 않고 약간의 지연을 둠
 - piggyback
 - 지연시간동안 상대방에게 보낼 데이터에 ACK 비트를 세트함으로 별도의 ACK를 생략
 - 지연 타이머는 500ms 이하이며 보통 50 ~ 200ms를 사용

TCP_MAXSEG

- 최대 세그먼트 크기의 지원은 모든 시스템이 지원하지 않음
 - 리눅스에서는 지원됨
- 최대 세그먼트 크기 변경은 연결 설정 이전에 함
- 너무 큰 값으로 변경되었을 시 연결설정 과정에서 지정값보다 작게 재지정됨

5.2 소켓 옵션 변경

5.2.1 소켓 옵션 변경 함수

accept()를 호출하기 전에 소켓 옵션을 지정

getsockopt()

소켓에 현재 지정되어 있는 옵션 값을 알아냄

setsockopt()

소켓 옵션을 변경함

사용방법

```
int getsockopt(int s, int level, int opt, const char *optval, int *optlen)
int setsockopt(int s, int level, int opt, const char *optval, int optlen)
```

소켓 옵션 변경 함수

■ 사용방법

- int s
 - 옵션 내용을 읽을 또는 변경할 소켓 번호
- int level
 - 프로토콜 레벨을 지정
 - 소켓 레벨의 옵션 : SOL_SOCKET
 - IP 프로토콜에 관한 옵션 : IPPROTO_IP
 - TCP에 관한 옵션 : IPPROTO_TCP
- int opt
 - 변경할 또는 읽을 옵션을 지정
- const char *optval
 - 지정하려는 또는 읽을 옵션 값을 가리키는 포인터
- int *optlen, int optlen
 - *optval의 크기를 나타냄
 - getsockopt()에서는 optlen을 setsockopt()에서는 *optlen을 사용

소켓 옵션의 종류

레벨	옵션	의미
SOL_SOCKET	SO_BROADCAST SO_DEBUG SO_REUSEADDR SO_LINGER SO_KEEPALIVE SO_OOBINLINE SO_RCVBUF SO_SNDBUF	방송형 메시지 전송 허용 DEBUG 모드를 선택 주소의 재사용 선택 소켓을 닫을 때 미전송된 데이터가 있어도 지정된 시간만큼 기다렸다가 소켓을 닫음 TCP의 keep-alive 동작 선택 OOB 데이터를 일반 데이터처럼 읽음 수신버퍼의 크기 변경 송신버퍼의 크기 변경
IPPROTO_IP	IP_TTL IP_MULTICAST_TTL IP_ADD_MEMBERSHIP IP_DROP_MEMBERSHIP IP_MULTICAST_LOOP IP_MULTICAST_IF	Time To Live 변경 멀티캐스트 데이터그램의 TTL 변경 멀티캐스트 그룹에 가입 멀티캐스트 그룹에서 탈퇴 멀티캐스트 데이터그램의 loopback 허용 여부 멀티캐스트 데이터그램 전송용 인터페이스 지정
IPPROTO_TCP	TCP_KEEPALIVE TCP_MAXSEG TCP_NODELAY	keep-alive 확인 메시지 전송 시간 지정 TCP의 MSS(최대 메시지 크기) 지정 Nagle 알고리즘의 선택

5.2.2 소켓 옵션 변경 예제 프로그램

소켓 옵션을 사용하여 수신버퍼의 크기 변경

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <unistd.h>
8
9 int main()
10 {
11     int s;
12     int value, length;
13
14     if((s = socket(AF_INET, SOCK_STREAM, 0)) < 0){
15         perror("소켓 실패");
16         exit(1);
17     }
```

- 14 ~ 17번 줄
 - 소켓 생성


```

18     length = sizeof(value);
19     if(getsockopt(s, SOL_SOCKET, SO_RCVBUF, &value, &length) < 0){
20         perror("소켓 실패");
21         exit(1);
22     }
23
24     printf("디폴트 수신 버퍼 크기: %dWn",value);
25     value = 1024;
26
27     setsockopt(s, SOL_SOCKET, SO_RCVBUF, &value, sizeof(value));
28     getsockopt(s, SOL_SOCKET, SO_RCVBUF, &value, &length);
29     printf("1024 로 변경한 수신버퍼 크기 %dWn",value);
30     return 0;
31 }

```

- 19 ~ 22번 줄
 - 수신 버퍼의 크기 값을 얻어옴

- 24 printf("디폴트 수신 버퍼 크기: %dWn",value);
 - 현재 디폴트로 설정된 수신 버퍼의 크기를 출력

- 27 setsockopt(s, SOL_SOCKET, SO_RCVBUF, &value, sizeof(value));
 - 수신 버퍼를 value 사이즈만큼 변경

- 28 getsockopt(s, SOL_SOCKET, SO_RCVBUF, &value, &length);
 - 변경된 수신 버퍼의 값을 얻어옴

실행화면

```
$ change_recvbuf
디폴트 수신 버퍼 크기: 87380
1024 로 변경한 수신버퍼 크기 2048
```

- 파일명 : change_recvbuf.c
- 컴파일 : gcc -o change_recvbuf change_recvbuf.c
- 사용법 : change_recvbuf
- 수신 버퍼의 크기는 커널이 두 배의 값으로 변경함
 - 1024로 설정하여도 출력에서는 2048이 나옴

6장 프로세스

- 6.1 프로세스의 이해
- 6.2 프로세스의 생성과 종료
- 6.3 데몬 서버 구축 방법

Overview

- 프로세스에 관한 기본 지식 습득
- 기초적인 프로세스 다루기
- 데몬 서버 구축 방법 습득

6.1 프로세스의 이해

6.1.1 프로세스

프로세스의 정의

현재 수행중인 프로그램

프로세스 식별자

프로세스를 구분하는 프로세스 등록번호

프로세스의 정의

- OS상에서 실행되는 개개의 프로그램
 - task라고도 함
 - 기계어로 이루어진 수동적인 프로그램이 아님
 - 시스템 자원을 할당받아 동작되고 있는 능동적인 프로그램
 - CPU의 레지스터, 프로그램 포인터, 스택 메모리 등 프로그램 실행에 필요한 자원을 할당 받음

프로세스 식별자

- 커널에서 관리하기 위해 내부 프로세스 테이블에 등록
 - 이 등록번호를 프로세스 식별자(PID)라 함
- 프로세스 식별자를 통해 Linux 내부의 모든 프로세스를 통제함
 - 프로세스 식별자를 통해 프로세스에게 신호를 전달
 - 생성과 동시에 발생하며 종료시 커널에 반환

6.1.2 프로세스 식별자 확인

getpid()

프로세스 ID 반환

getpgrp()

프로세스의 그룹 ID 반환

getppid()

부모 프로세스 ID 반환

getpgid()

부모 프로세스의 그룹 ID 반환

getpid()

- 호출 프로세스의 프로세스 ID를 반환
- 입력 값은 없음

getpgrp()

- 호출 프로세스의 그룹 ID 반환
- 입력 값은 없음

getppid()

- 부모 프로세스 ID 반환

getpgid()

- 프로세스 ID를 입력 값으로 가짐
- 성공시 부모 프로세스의 그룹 ID 반환
- 에러시 pid - 1을 반환

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     int pid;
8
9     pid = fork();
10
11     if(pid != 0)
12     {
13         printf("부모 프로세스의 pid = %d\n", getpid());
14         printf("부모 프로세스의 ppid = %d\n", getppid());
15         printf("부모 프로세스의 pgrd = %d\n", getpgrp());
16         printf("부모 프로세스의 pgid = %d\n", getpgid(getpid()));
17         printf("Wn");
18     }
19     else
20     {
21         printf("자식 프로세스의 pid = %d\n", getpid());
22         printf("자식 프로세스의 ppid = %d\n", getppid());
23         printf("자식 프로세스의 pgrd = %d\n", getpgrp());
24         printf("자식 프로세스의 pgid = %d\n", getpgid(getpid()));
25         printf("Wn");
26     }
27     exit(0);
28 }

```

실행화면

```
$ get_pid
자식 프로세스의 pid = 11262
자식 프로세스의 ppid = 11261
자식 프로세스의 pgrd = 11261
자식 프로세스의 pgid = 11261

부모 프로세스의 pid = 11261
부모 프로세스의 ppid = 11147
부모 프로세스의 pgrd = 11261
부모 프로세스의 pgid = 11261
```

실행

- 파일명 : get_pid.c
- 컴파일 : gcc -o get_pid get_pid.c
- 사용법 : get_pid

프로세스 그룹

여러 프로세스들이 모여서 하나의 프로세스 그룹을 형성

프로세스 그룹

- 어떤 하나의 작업을 수행하는데 관계된 프로세스들의 집합
- 프로세스 그룹 ID
 - 동일 프로세스 그룹 내에 있는 프로세스들은 동일한 프로세스 그룹 ID를 소유
- 프로세스 그룹 생존시간
 - 프로세스 그룹의 생성에서 모든 프로세스의 종료까지의 시간 간격
 - 프로세스 그룹 리더에 의해 프로세스에서의 프로세스를 생성, 종료
 - 같은 그룹내의 모든 프로세스가 없어질 때까지 그 그룹은 존재함

6.2 프로세스의 생성과 종료

6.2.1 프로세스의 생성

fork()

프로세스의 생성 함수

사용문법

```
1 int PID = fork();
2 if (PID == 0)
3 {
4     child_work();
5 }
6 else if (PID > 0)
7 {
8     parent_work();
9 }
10 else
11 {
12     error();
13 }
```

fork()

- 리눅스에서 유일하게 프로세스를 생성하는 함수
- 성공시 프로세스 ID를 반환함
- 동작방법
 - fork()를 실행한 프로세스로부터 새 프로세스가 복제됨
 - fork()를 실행한 프로세스는 부모 프로세스라 함
 - 새 프로세스는 자식 프로세스라 함
- 사용문법
 - 입력값은 없음
 - 4 child_work();
 - 자식 프로세스용 코드
 - 8 parent_work();
 - 부모 프로세스용 코드
 - 12 error();
 - fork() 에러

6.2.2 프로세스의 종료

exit()

정상적인 종료에서 사용

abort()

비정상적인 종료에서 사용

exit()

- 정상적인 종료
 - main() 함수의 묵시적인 종료
 - main() 함수의 return 사용
 - exit() 호출
- 사용방법
 - 종료의 상태로 사용할 정수 값을 인자로 받아 해당 상태로 종료됨

abort()

- 비정상적인 종료
 - 신호에 의한 종료
 - abort() 호출

6.3 데몬 서버 구축 방법

6.3.1 데몬 프로세스

데몬 프로세스 생성

데몬은 백그라운드로 실행되는 프로세스로 특정 터미널 제어와 관계없이 실행되는 프로세스

데몬 프로세스 생성

- 백그라운드로 실행되는 프로세스
 - 터미널에서 데몬 실행 후 사용자가 로그아웃 해도 데몬은 종료되지 않음
 - 데몬은 kill 명령이나 SIGKILL 시그널을 보냄으로 종료

사용문법

```
1 struct sigaction sact;
2 sigset_t mask;
3
4 if((pid = fork()) != 0)
5     exit(0);
6
7 setsid();
8
9 sact.sa_handler = SIG_IGN;
10 sact.sa_flags = 0;
11 sigemptyset(&sact.sa_mask);
12 sigaddset(&sact.sa_mask, SIGHUP);
13 sigaction(SIGHUP, &sact);
14
15 if((pid = fork()) != 0)
16     exit(0);
17
18 chdir("/");
19 umask(0);
20 for(i=0 ; i<MAXFD ; i++)
21     close(1);
```

사용문법

- 위의 코드를 프로그램 앞부분에 추가
- 4 ~ 5번 줄
 - 부모 프로세스를 종료시키고 자식 프로세스에서 실행됨
- 7 setsid();
 - 스스로 세션 리더가 됨
- 9 ~ 13번 줄
 - SIGHUP 시그널을 무시
 - 손자 프로세스와 터미널의 연관을 끊음
- 15 ~ 16번 줄
 - 손자 프로세스를 생성하고 부모 프로세스를 종료
- 18 chdir("/");
 - 작업 디렉토리를 루트 디렉토리로 변경
- 19 umask(0);
 - 새로 생성되는 파일이 임의의 소유 권한을 가지게 함
- 20 ~ 21번 줄
 - 혹시 개설되어 있을 소켓을 닫음

6.3.2 데몬 서버 종류

독립형 데몬 서버

httpd, sendmail, named 등과 같이 항상 실행되고 있으면서 서비스를 제공하는 데몬

독립형 데몬 서버 구축

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5 #include <signal.h>
6 #include <syslog.h>
7 #include <stdarg.h>
8 #define BUF_LEN 128
9 #define MAXFD 64
10
11 int main(int argc, char *argv[])
12 {
13     struct sockaddr_in server_addr, client_addr;
14     int server_fd, client_fd;
15     int i, len, len_out;
16     pid_t pid;
17     char buf[BUF_LEN+1];
18 }
```

- 14 int server_fd, client_fd;
 - 소켓 번호를 저장하는 변수

```

19  if(argc != 2)
20  {
21      printf("사용법: %s portWn", argv[0]);
22      exit(0);
23  }
24
25  if((pid = fork()) != 0)
26      exit(0);
27
28  setsid();
29  signal(SIGHUP, SIG_IGN);
30
31  if((pid = fork()) != 0)
32      exit(0);
33
34  chdir ("/");
35  umask(0);
36
37  for(i = 0 ; i < MAXFD ; i++)
38      close(i);
39
40  if((server_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
41      exit(0);
42

```

- 25 ~ 26번 줄
 - 서버 프로그램을 데몬 서버로 만듦
 - 26 exit(0);
 - 부모 프로세스를 종료시키고 자식 프로세스에서 실행됨
- 28 setsid();
 - 세션리더로 만듦
- 29 signal(SIGHUP, SIG_IGN);
 - SIGHUP 시그널을 무시함
- 31 ~ 32번 줄
 - 손자 프로세스를 생성하고 부모 프로세스를 종료
- 34 chdir ("/");
 - 작업 디렉토리를 루트 디렉토리로 변경
- 35 umask(0);
 - 새로 생성되는 파일이 임의의 소유 권한을 가지게 함
- 40 ~ 41번 줄
 - 소켓 생성

```

43     bzero((char *)&server_addr, sizeof(server_addr));
44     server_addr.sin_family = AF_INET;
45     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
46     server_addr.sin_port = htons(atoi(argv[1]));
47
48     if(bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
49         exit(0);
50
51     listen(server_fd, 5);
52     while(1)
53     {
54         len = sizeof(client_addr);
55
56         if((client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &len)) < 0)
57             exit(0);
58
59         if((len_out = read(client_fd, buf, sizeof(buf))) < 0)
60             exit(0);
61
62         write(client_fd, buf, len_out);
63         close(client_fd);
64     }
65     close(server_fd);
66 }

```

- 43 `bzero((char *)&server_addr, sizeof(server_addr));`
 - `server_addr` 을 'W0' 으로 초기화
- 44 ~ 46번 줄
 - `server_addr` 셋팅
- 48 ~ 49번 줄
 - `bind()` 호출
- 51 `listen(server_fd, 5);`
 - 소켓을 수동 대기모드로 셋팅
- 52 ~ 64번 줄
 - interactive echo 서비스 수행
- 56 ~ 57번 줄
 - 연결 요청을 기다림

실행화면

서버 실행 확인

```
$ ps -x
  PID TTY          STAT       TIME COMMAND
18965 ?            S           0:00 myecho_daemon 5000
28396 ?            S           0:00 [sshd]
28397 pts/2        S           0:00 -bash
28521 pts/2        R           0:00 ps -x
```

클라이언트 실행

```
$ tcp_echocli 210.115.49.220
Input any string : daemon server test
Echoed string : daemon server test
```

실행

- 파일명 : echoserv_daemon.c
- 컴파일 : gcc -o echoserv_daemon echoserv_daemon.c
- 사용법
 - 데몬 서버 : echoserv_daemon 포트번호
 - 클라이언트 : tcp_echocli 서버IP
- 클라이언트 프로그램은 3장의 tcp_chatcli.c를 사용

7장 시그널

7.1 시그널 종류

7.2 시그널 처리

7.3 SIGCHLD와 프로세스 종료

Overview

- 시그널의 종류 습득
- 시그널의 처리 방법 습득
- SIGCHLD 시그널 처리 방법 습득
- 자식 프로세스 종료시의 부모 프로세스에서 이를 처리하는 방법 습득

7.1 시그널 종류

7.1.1 시그널의 종류

시그널의 정의

프로세스에서 어떤 이벤트의 발생을 다른 프로세스에게 알리는 도구

시그널의 발생

kill() 시스템 콜을 사용

```
int kill(pid_t pid, int sig);
```

프로세스가 자신에게 시그널을 보냄

```
int raise(int sig);
```

시그널의 정의

- 소프트웨어 인터럽트
 - 운영체제 또는 커널에서 일반 프로세스로 보냄
 - 일반 프로세스에서 커널의 도움을 받아 다른 프로세스로 시그널을 넘김

시그널의 발생

- kill() 시스템 콜
 - 다른 프로세스에게 시그널이 발생되도록 함
 - 프로세스 pid인 프로세스에게 sig 시그널이 발생
- raise()
 - 프로세스가 자신에게 시그널을 보냄

시그널의 종류

시그널	발생 조건
SIGINT	· 인터럽트키(CTRL-C)를 입력했을 때 발생
SIGKILL	· 강제 종료 시그널로서, 프로세스에서 이 시그널을 무시하거나 블록할 수 없음
SIGIO	· 비동기 입출력이 발생했을 때 전달
SIGPIPE	· 닫힌 파이프나 소켓에 데이터르 쓰거나 읽기를 시도할 때 발생
SIGCHLD	· 프로세스가 종료되거나 취소될 때 부모 프로세스에게 전달
SIGPWR	· 전원의 중단 및 재시작 시에 init 프로세스로 전달
SIGTSTP	· 사용자가 키보드에서 중지키(CTRL-Z)를 눌렀을 때 발생
SIGSYS	· 잘못된 시스템 호출 시에 발생
SIGURG	· 대역외 데이터를 수신 시에 발생
SIGUSR1	· 사용자가 임의의 목적으로 사용할 수 있는 시그널
SIGUSR2	
SIGHUP	· 터미널과 연결이 끊어졌을 때 세션 리더에게 보내짐
SIGQUIT	· 종료키(CTRL-W)를 눌렀을 때 전달
SIGILL	· 프로세스가 규칙에 어긋난 명령을 수행하려고 할 때 전달
SIGTRAP	· 프로그램이 디버깅 지점에 도달하면 전달
SIGABRT	· abort() 함수를 호출하면 발생
SIGFPE	· 숫자를 0으로 나누거나 연산 에러 시 발생
SIGVTALRM	· setitimer() 함수에 의한 가상 타이머 시간 만료를 알림

시그널의 종류

- 리눅스 시그널의 이름은 SIG로 시작
- signal.h 파일에 소개되어 있음

7.2 시그널 처리

7.2.1 시그널 처리 기본 동작

정해진 기본 동작 수행

프로세스 종료 또는 시그널 무시

사용자가 지정한 작업 수행

시그널 핸들러 수행, 시그널 무시, 시그널 블록

정해진 기본 동작 수행

- 프로세스가 시그널을 수신시 각 시그널에 따라 처리할 기본 동작이 미리 정해져 있음
 - 대부분 프로세스 종료임
 - SIGKILL, SIGSTOP 시그널
 - 기본 동작으로 프로세스 종료임
 - 무시나 시그널 핸들러 등록이 허용되지 않음

사용자가 지정한 작업 수행

- 시그널 핸들러 수행
 - 미리 정해진 함수를 수행
- 시그널 무시
 - 프로세스는 시그널을 무시하므로 아무 영향이 없음
- 시그널 블록
 - 해당 시그널의 수신시 시그널 대기큐에 들어감
 - 현재의 동작을 계속하고 나중에 시그널을 처리
 - bitmask 형태의 시그널 블록 마스크에 추가해서 블록함
 - 블록된 시그널은 중복되지 않음

7.2.2 시그널 핸들러

시그널 집합

여러 개의 시그널을 한 번에 표시하기 위해 sigset_t를 사용

sigset_t 타입의 변수를 다루는 함수

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

시그널 집합

- sigset_t 변수에는 여러 개의 시그널을 bitmask 형태로 누적하여 표현
- sigemptyset()
 - 시그널 집합을 비어 있는 상태로 초기화
- sigfillset()
 - 시스템에 정의되어 있는 모든 시그널들을 시그널 집합에 넣음
- sigaddset()
 - 시그널 집합에 특정 시그널을 추가
- sigdelset()
 - 시그널 집합에서 특정 시그널을 제거
- sigismember()
 - 시그널 집합 안에 특정 시그널이 존재하는지 여부를 확인

시그널 집합 함수의 사용

```
1 sigset_t signals;
2
3 sigemptyset(&signals);
4 sigaddset(&signals, SIGINT);
5
6 sigfillset(&signals);
7 sigdelset(&signals, SIGINT);
8
9 if(sigismember(&signals, SIGINT))
10 {
11     ...
12 }
```

시그널 집합 함수의 사용

- 1 sigset_t signals;
 - sigset_t 타입의 시그널 집합 signals를 생성
- 3 sigemptyset(&signals);
 - 시그널 집합 signals를 비어 있는 상태로 만듦
- 4 sigaddset(&signals, SIGINT);
 - 시그널 집합 signals에 SIGINT를 추가
- 6 sigfillset(&signals);
 - 시그널 집합 signals를 fill 상태로 만듦
- 7 sigdelset(&signals, SIGINT);
 - 시그널 집합 signals에서 SIGINT 시그널 제거
- 9 if(sigismember(&signals, SIGINT))
 - 시그널 집합 signals에서 SIGINT가 있는지 확인하는 if문

시그널 핸들러 설정

시그널 핸들러를 등록하기 위해 `sigaction()` 함수를 사용

`sigaction()` 함수

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

시그널 핸들러 설정

- `sigaction()` 함수
 - 인자 `signum`
 - 처리할 대상이 되는 시그널
 - SIGKILL과 SIGSTOP를 제외한 모든 시그널이 가능함
 - 인자 `act`
 - `signum` 시그널 발생시 동작을 결정에 참조되는 구조체
 - 시그널 핸들러 이름, 실행도중 블록시킬 시그널 집합, 환경설정용 `flag`인자 포함

sigaction 구조체

```
struct sigaction{
    void(*sa_handler)(int);
    void(*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void(*sa_restore)(void);
}
```

sigaction 구조체

- sa_handler, sa_sigaction
 - 시그널 핸들러를 가리킴
 - 일반적으로 sa_handler를 사용하고 sa_sigaction은 부가적인 정보를 이용할 때 사용

- sa_mask
 - 시그널 핸들러가 동작되는 동안 블록시킬 시그널 집합
 - 시그널 핸들러가 실행되는 도중에 sa_mask 없는 시그널 도착시
 - 시그널 핸들러는 인터럽트 되고 새로 도착한 시그널에 대한 처리가 진행됨
 - 인터럽트를 받지 않는게 안전하므로 sigfillset(&act.sa_mask)로 모두 포함시킴

- sa_restore
 - 사용되지 않음

■ sa_flags

- SA_RESTART
 - 인터럽트 되었던 시스템 콜이 시그널 핸들러 처리한 후에 계속 수행되도록 함
- SA_NOCLDWAIT
 - 좀비 프로세스를 만들지 않도록 하기 위한 플래그
 - SIGCHLD 시그널 처리시 주로 사용됨
- SA_NODEFER
 - 도착한 시그널의 수만큼 핸들러가 수행됨
- SA_SIGINFO
 - 미설정시 시그널이 발생하면 sa_handler에 명시된 핸들러 함수가 호출됨
 - 설정시 시그널이 발생하면 sa_sigaction에 명시된 핸들러 함수가 호출됨
- SA_ONESHOT or SA_RESETHAND
 - 시그널 핸들러가 한 번 실행된 후에 시그널 처리 기본 동작으로 되돌아감
 - 설치한 시그널 핸들러는 한 번만 실행되고 이후의 시그널은 SIG_DFL에 의해 수행

signal_unsafe 함수

시그널 핸들러 내에서 호출하면 그 결과가 불확실한 함수

signal() 함수

sigaction()과 유사한 시스템 콜

signal_unsafe 함수

- 결과를 예측할 수 없는 함수
 - malloc(), free(), alarm(), sleep(), fcntl(), fork() 등
 - 시그널 핸들러 외부의 작업과 충돌을 일으킬 가능성이 있음
 - 정적인 변수나 전역 변수를 사용
- 호출해야 할 경우
 - 직접 호출보다 핸들러 내에서는 특정 플래그만 설정
 - 핸들러가 종료된 후에 플래그를 확인하고 signal_unsafe한 함수를 핸들러 밖에서 호출

signal() 함수

- sigaction()이 소개되기 이전에 사용되던 함수
- 시그널 핸들러를 등록하거나 시그널 무시를 설정함
- 현재 사용하지 않기를 권장함

7.2.3 시그널 처리 예

SIGINT 처리 예

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4 #include<unistd.h>
5 #include<errno.h>
6 #include<signal.h>
7
8 void catch_sigint(int signum);
9 int count;
10
11 int main(int argc, char *argv[])
12 {
13     struct sigaction act;
14     sigset_t masksets;
15     int i;
16     char buf[10];
17     sigfillset(&masksets);
18
```

```

19  act.sa_handler = catch_sigint;
20
21  act.sa_mask = masksets;
22  act.sa_flags = 0;
23  sigaction(SIGINT, &act, NULL);
24
25  for(count=1 ; count<5 ; count++)
26      read(0, buf, sizeof(buf));
27
28  return 0;
29 }
30
31 void catch_sigint(int signum)
32 {
33     printf("\n CTRL-C가 %d 번 눌렀습니다. \n", count);
34 }

```

- 19 act.sa_handler = catch_sigint;

- 21 act.sa_mask = masksets;
 - 시그널 핸들러가 실행되는 동안 모든 시그널을 블록함

- 31 ~ 34번 줄
 - 시그널 핸들러

실행화면

```
$ catch_SIGINT  
  
CTRL-C가 1 번 눌렀습니다.  
CTRL-C가 2 번 눌렀습니다.  
CTRL-C가 3 번 눌렀습니다.  
CTRL-C가 4 번 눌렀습니다.
```

실행

- 파일명 : catch_SIGINT.c
- 컴파일 : gcc -o catch_SIGINT catch_SIGINT.c
- 사용법 : catch_SIGINT
- 동작설명
 - CTRL-C를 눌러도 catch_sigint()가 SIGINT를 잡아서 처리하므로 종료가 되지 않음
 - 시그널 핸들러를 수행하는 동안 도착한 모든 시그널은 블록됨

7.3 SIGCHLD와 프로세스의 종료

7.3.1 프로세스의 종료

SIGCHLD 시그널

프로세스 종료 시 자신의 부모 프로세스에게 보내는 시그널

wait()

부모 프로세스가 자식 프로세스의 종료 시점이나 종료 상태 값을 알기 위해 사용

```
pid_t pid;  
int stat;  
pid = wait(&stat);
```

SIGCHLD 시그널

- 부모 프로세스는 원하면 이 시그널을 받아 처리함

wait()

- 부모 프로세스가 자식 프로세스가 종료된 것을 확인하고 다른 작업을 할 경우
 - pid = wait(&stat); 의 작업 수행
 - 자식 프로세스가 종료될 때까지 블록됨
 - 종료 상태 값은 stat 인자로 리턴됨
 - 위의 작업 이후에 다른 작업 수행됨
- 좀비 상태 발생
 - 자식 프로세스 종료 후에 부모 프로세스가 wait() 호출로 종료 상태를 읽지 않을 경우 발생
 - 부모 프로세스가 나중에라도 자식 프로세스의 종료 상태를 읽기 위해 좀비 상태를 만듦
 - 좀비 상태의 프로세스는 사용자 영역의 메모리는 모두 free임
 - 커널이 관리하는 메모리는 남아 있으므로 메모리 낭비가 됨
- SIGCHLD를 잡아 처리하는 것으로 wait()를 대신할 수 있음
 - 인터럽트 방식으로 자식 프로세스의 종료 시점 파악

기본동작으로서 SIGCHLD를 무시하는 경우

부모 프로세스가 자식 프로세스의 종료 상태를 읽을 필요가 없을 때

기본동작으로서 SIGCHLD를 무시하는 경우

- 부모 프로세스가 자식 프로세스의 종료 상태를 읽을 필요가 없을 때 단순히 무시
- 부모 프로세스가 자식 프로세스의 종료 시점을 SIGCHLD를 통해 파악할 경우
 - 부모 프로세스에서 wait()를 호출한 이후에 자식 프로세스가 종료
 - 부모 프로세스는 wait()함수에서 대기
 - 자식 프로세스가 종료할 때 wait()함수가 리턴
 - 부모 프로세스가 wait()를 호출하기 이전에 자식 프로세스가 종료
 - 자식 프로세스가 좀비가 됨
 - 부모 프로세스는 필요시 나중에 wait()를 호출하여 자식의 종료 상태를 읽음
 - 부모 프로세스가 wait()를 호출하지 않고 종료
 - 시스템의 init 프로세스가 좀비 프로세스에 대해 부모 프로세스 역할을 함
 - 대신 wait()를 호출

SIG_IGN으로 SIGCHLD를 무시하는 경우

시그널 핸들러에 SIG_IGN을 등록하여 SIGCHLD를 무시

시그널 핸들러를 등록한 경우

SIGCHLD 시그널 핸들러를 등록하고 wait() 함수로 자식의 종료 상태를 얻음

SIG_IGN으로 SIGCHLD를 무시하는 경우

- 기본 동작으로 SIGCHLD를 무시하는 것과 거의 같은 동작을 함
 - 부모 프로세스가 wait()호출 전에 자식 프로세스가 종료되어도 좀비가 발생하지 않음
 - 이미 종료된 자식 프로세스의 상태를 얻지 못함
 - wait()호출 시 자식 프로세스가 없으면 -1을 리턴

시그널 핸들러를 등록한 경우

- 부모 프로세스가 자신의 작업을 수행하다가 자식 프로세스의 종료 시점에 종료 상태를 알고자 할 때
 - 미리 wait()를 호출하여 블록될 필요가 없음
 - 시그널 핸들러가 실행되어도 자식 프로세스의 좀비 상태는 끝나지 않음
 - wait()를 실행해야 함

7.3.2 프로세스의 종료 처리

wait() 함수의 동작 이해

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <signal.h>
6 #include <sys/types.h>
7 #include <sys/wait.h>
8 #include <errno.h>
9
10 void catch_sigchld(int signo)
11 {
12     puts("[ 부모 프로세스 ] catch SIGCHLD");
13 }
14
15 int child_stat;
16
17 int main(int argc, char *argv[])
18 {
19     int i, n;
20     struct sigaction sact;
21     sact.sa_flags=0;
22     sigemptyset(&sact.sa_mask);
23     sigaddset(&sact.sa_mask, SIGCHLD);
24 }
```

- 10 ~ 13번 줄
 - 사용자 시그널 핸들러 함수

- 15 int child_stat;
 - 종료 상태 값을 저장할 변수

```

25     sact.sa_handler = SIG_DFL;
26     sigaction(SIGCHLD, &sact, NULL);
27
28     for(i=0 ; i<5 ; i++)
29     {
30         if(fork()==0)
31         {
32             if(i > 2)
33                 sleep(5);
34
35             printf(" [ %d번 자식 프로세스 ] , PID=%d, PPID=%d 종료됨\n", i, getpid(), getppid());
36             exit(13);
37         }
38     }
39
40     sleep(3);
41     puts("##### 프로세스 현황 #####");
42     system("ps -a");
43     puts("#####");
44
45     puts(" [ 부모 프로세스 wait 시스템 콜 호출 ]");

```

- 25 `sact.sa_handler = SIG_DFL;`
 - 시그널 처리 등록 옵션
 - 기본 동작으로 시그널 무시

- 28 ~ 38번 줄
 - 5개의 프로세스를 생성
 - 0, 1, 2번 프로세스는 즉시 종료
 - 3, 4번 프로세스는 5초간 `sleep()`한 후에 종료
 - 부모의 `wait()` 호출보다 늦게 종료

- 36 `exit(13);`
 - 종료값 13번

```

46  while(1)
47  {
48      child_stat = -1;
49      n = wait(&child_stat);
50      printf("wait = %d (child process stat=%d)\n", n, child_stat);
51      if( n == -1)
52      {
53          if(errno == ECHILD)
54          {
55              perror("기다릴 자식프로세스가 없음");
56              break;
57          }
58          else if(errno == EINTR)
59          {
60              perror("wait 시스템 콜이 인터럽트 됨");
61              continue;
62          }
63      }
64  }
65  puts("[ 부모 프로세스 종료 ]");
66  return 0;
67 }

```

- 48 child_stat = -1;
- child_stat값 초기화

실행화면

```

$ test_wait
[ 0번 자식 프로세스 ] , PID=21376, PPID=21375 종료됨
[ 1번 자식 프로세스 ] , PID=21377, PPID=21375 종료됨
[ 2번 자식 프로세스 ] , PID=21378, PPID=21375 종료됨
##### 프로세스 현황 #####
  PID TTY          TIME CMD
21375 pts/2        00:00:00 test_wait
21376 pts/2        00:00:00 test_wait <defunct>
21377 pts/2        00:00:00 test_wait <defunct>
21378 pts/2        00:00:00 test_wait <defunct>
21379 pts/2        00:00:00 test_wait
21380 pts/2        00:00:00 test_wait
21383 pts/2        00:00:00 ps
#####
[ 부모 프로세스 wait 시스템 콜 호출 ]
wait = 21376 (child process stat=3328)
wait = 21377 (child process stat=3328)
wait = 21378 (child process stat=3328)
[ 3번 자식 프로세스 ] , PID=21379, PPID=21375 종료됨
[ 4번 자식 프로세스 ] , PID=21380, PPID=21375 종료됨
wait = 21379 (child process stat=3328)
wait = 21380 (child process stat=3328)
wait = -1 (child process stat=-1)
기다릴 자식프로세스가 없음: No child processes
[ 부모 프로세스 종료 ]

```

실행

- 파일명 : wait_test.c
- 컴파일 : gcc -o wait_test wait_test.c
- 사용법 : wait_test
- 동작 설명
 - 5개의 프로세스 중 wait() 호출 이전에 종료한 0, 1, 2는 좀비가 됨

시그널 무시를 등록한 경우

wait.c 의 25번째 줄을 `sact.sa_handler = SIG_IGN;` 으로 바꿈

```
$ wait SIGIGN
[ 0번 자식 프로세스 ] , PID=21398, PPID=21397 종료됨
[ 1번 자식 프로세스 ] , PID=21399, PPID=21397 종료됨
[ 2번 자식 프로세스 ] , PID=21400, PPID=21397 종료됨
##### 프로세스 현황 #####
  PID TTY          TIME CMD
21397 pts/2        00:00:00 wait_SIGIGN
21401 pts/2        00:00:00 wait_SIGIGN
21402 pts/2        00:00:00 wait_SIGIGN
21403 pts/2        00:00:00 ps
#####
[ 부모 프로세스 wait 시스템 콜 호출 ]
[ 3번 자식 프로세스 ] , PID=21401, PPID=21397 종료됨
wait = 21401 (child process stat=3328)
[ 4번 자식 프로세스 ] , PID=21402, PPID=21397 종료됨
wait = 21402 (child process stat=3328)
wait = -1 (child process stat=-1)
기다릴 자식 프로세스가 없음: No child processes
[ 부모 프로세스 종료 ]
```

실행

- 파일명 : wait_SIGIGN.c
- 컴파일 : gcc -o wait_SIGIGN wait_SIGIGN.c
- 사용법 : wait_SIGIGN
- 동작 설명
 - 좀비 프로세스가 발생하지 않음
 - 0, 1, 2번 프로세스의 종료 상태는 영원히 알 수 없음

SIGCHLD 시그널 핸들러를 등록한 경우

wait.c 의 25번째 줄을 `sact.sa_handler = catch_sigchld;` 로 바꿈

```
$ wait_SIGCHLD
[ 0번 자식 프로세스 ] , PID=21410, PPID=21409 종료됨
[ 부모 프로세스 ] catch SIGCHLD
[ 1번 자식 프로세스 ] , PID=21411, PPID=21409 종료됨
[ 부모 프로세스 ] catch SIGCHLD
[ 2번 자식 프로세스 ] , PID=21412, PPID=21409 종료됨
[ 부모 프로세스 ] catch SIGCHLD
##### 프로세스 현황 #####
  PID TTY          TIME CMD
21409 pts/2        00:00:00 wait_SIGCHLD
21410 pts/2        00:00:00 wait_SIGCHLD <defunct>
21411 pts/2        00:00:00 wait_SIGCHLD <defunct>
21412 pts/2        00:00:00 wait_SIGCHLD <defunct>
21413 pts/2        00:00:00 wait_SIGCHLD
21414 pts/2        00:00:00 wait_SIGCHLD
21415 pts/2        00:00:00 ps
[ 부모 프로세스 ] catch SIGCHLD
#####
[ 부모 프로세스 wait 시스템 콜 호출 ]
wait = 21410 (child process stat=3328)
wait = 21411 (child process stat=3328)
wait = 21412 (child process stat=3328)
[ 3번 자식 프로세스 ] , PID=21413, PPID=21409 종료됨
[ 부모 프로세스 ] catch SIGCHLD
wait = 21413 (child process stat=3328)
[ 4번 자식 프로세스 ] , PID=21414, PPID=21409 종료됨
[ 부모 프로세스 ] catch SIGCHLD
wait = 21414 (child process stat=3328)
wait = -1 (child process stat=-1)
기다릴 자식프로세스가 없음: No child processes
[ 부모 프로세스 종료 ]
```

실행

- 파일명 : wait_SIGCHLD.c
- 컴파일 : `gcc -o wait_SIGCHLD wait_SIGCHLD.c`
- 사용법 : `wait_SIGCHLD`
- 동작 설명
 - 0, 1, 2번 프로세스가 종료하면서 발생한 SIGCHLD에 의해 시그널 핸들러 `catch_sigchld()` 호출
 - `catch SIGCHLD` 문장이 출력됨
 - 부모 프로세스에게 아직 종료하지 않은 자식 프로세스가 있으므로 이후 `wait()`로 대기
 - 이 상태에서 자식 프로세스가 종료하면 부모 프로세스에게 SIGCHLD 시그널 전달

시스템 콜 인터럽트 경우

부모 프로세스가 다른 시스템 콜을 호출한 상태에서 SIGCHLD 시그널 도착한 경우

waitpid()

특정 PID를 갖는 자식 프로세스의 종료만을 기다림

시스템 콜 인터럽트 경우

- SIGCHLD의 시그널 핸들러 함수를 등록했다면 SIGCHLD 시스템 콜을 인터럽트 시킴
- 시그널 처리가 기본 동작이나 시그널 무시 상태이면 시스템 콜은 아무 영향 받지 않음

waitpid()

- 만약 앞으로 종료할 자식 프로세스가 없을 경우
 - 부모 프로세스는 영원히 기다리게 됨
 - flags에서 WNOHANG 옵션을 사용해서 문제 해결
 - waitpid() 호출 시에 실행중인 자식 프로세스가 있을 때에만 블록
 - 실행중이거나 종료된 자식 프로세스가 없으면 블록되지 않음

8장

프로세스간 통신

- 8.1 파이프
- 8.2 FIFO
- 8.3 메시지큐
- 8.4 공유메모리
- 8.5 세마포어

Overview

- 프로세스간 통신 기술로의 파이프, 메시지큐, 세마포어, 공유메모리의 기술 습득
- 각 기술들을 통신 프로그램에서 이용하는 방법 익힘

8.1 파이프

파이프의 정의

운영체제가 제공하는 프로세스간 통신 채널로서 특별한 타입의 파일

8.1.1 파이프 생성

pipe()

```
#include <unistd.h>
int pipe(int fd[2]);
```

파이프의 정의

- 운영체제가 관리하는 임시 파일
 - 일반 파일과 달리 메모리에 저장되지 않음
 - 데이터 저장용이 아님
 - 프로세스간 데이터 전달용으로 사용
- 통신 기능
 - 송신 프로세스에서 파이프에 데이터를 쓰고 수신 프로세스에서 파이프에서 데이터를 읽음
 - 스트림 채널을 제공하여 송신된 데이터는 바이트 순서가 유지됨
 - 같은 컴퓨터 내의 프로세스 간에 스트림 채널을 제공

pipe()

- 파이프를 열기 위한 명령
- 두 개의 파일 디스크립터가 생성됨
 - 0번 : 파이프로부터 데이터를 읽음
 - 1번 : 파이프에 데이터를 씀

fork() 호출

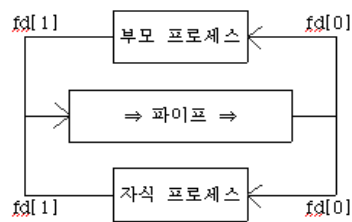


그림 8.1 파이프 생성후 fork()를 호출

불필요한 파일 디스크립터 해제

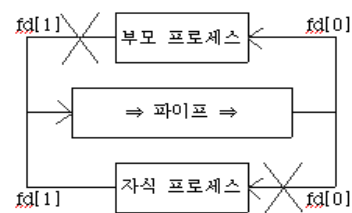


그림 8.2 사용하지 않는 파일 디스크립터를 닫음

fork() 호출

- 파일 디스크립터 `fd[0]`, `fd[1]`
 - 부모와 자식 프로세스는 모두 파일 디스크립터 `fd[0]`로부터 데이터를 읽음
 - 부모와 자식 프로세스는 모두 파일 디스크립터 `fd[1]`로 데이터를 씀

불필요한 파일 디스크립터 해제

- 자식 프로세스가 부모 프로세스로 데이터를 보내는 경우
 - 자식 프로세스는 `fd[1]`로 데이터를 쓰고 부모 프로세스는 `fd[0]`으로 읽음
 - 자식 프로세스의 `fd[0]`과 부모 프로세스의 `fd[1]`은 사용하지 않으므로 닫음

소스로 구현

```
1 int fd[2];
2 pid_t pid;
3 pipe(fd);
4 if((pid=fork()) < 0)
5 {
6     exit(0);
7 }
8 else if(pid > 0)
9 {
10     close(fd[1]);
11 }
12 else if(pid == 0)
13 {
14     close(fd[0]);
15 }
```

소스로 구현

- 4 ~ 7번 줄
 - 자식 프로세스를 생성

- 8 ~ 11번 줄
 - 부모 프로세서에서 쓰기용 파일 디스크립터를 제거

- 12 ~ 15번 줄
 - 자식 프로세서에서 읽기용 파일 디스크립터를 제거

- 단방향 스트림 채널만 제공
 - 양방향 통신을 하려면 파이프를 추가 생성해야함

8.1.2 파이프를 이용한 에코 서버 프로그램

에코 서버

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <errno.h>
7 #include <sys/time.h>
8 #include <sys/socket.h>
9 #include <sys/types.h>
10
11 #define MAX_BUFFER 512
12
13 typedef struct message {
14     struct sockaddr_in addr;
15     char data[MAX_BUFFER];
16 }mesg_t;
17
18 void child_process(int sock,int pipefd[])
19 {
20     mesg_t pmsg;
21     int nbytes=0,
22     length = sizeof(struct sockaddr);
23     close(pipefd[1]);
24 }
```

- 13 ~ 16번 줄
 - 파이프에 쓰는 데이터 구조

- 14 struct sockaddr_in addr;
 - 클라이언트 주소

- 15 char data[MAX_BUFFER];
 - 에코할 데이터 저장소

- 18 ~ 35번 줄
 - 자식 프로세스

```

25     while(1)
26     {
27         nbytes = read(pipefd[0], (char *)&pmsg, sizeof(mesg_t));
28         if(nbytes < 0)
29             errquit("읽기 실패");
30
31         printf("자식 프로세스 : 파이프로부터 읽음 %d\n", nbytes);
32         nbytes = sendto(sock, &pmsg.data, strlen(pmsg.data), 0,
                        (struct sockaddr*)&pmsg.addr, length);
33         printf("자식 프로세스 : %d 바이트의 echo가 응답됨\n", nbytes);
34     }
35 }
36
37 void parent_process(int sock, int pipefd[])
38 {
39     mesg_t pmsg;
40     int nbytes, length=sizeof(struct sockaddr);
41     close(pipefd[0]);
42
43     printf("대기 중...\n");
44     while(1)
45     {
46         nbytes = recvfrom(sock, (void*)&pmsg.data, MAX_BUFFER, 0,
                        (struct sockaddr*)&pmsg.addr, &length);
47
48         if(nbytes < 0)
49             errquit("소켓에서 읽어오기 실패");

```

■ 25 ~ 34번 줄

- 읽기 대기

■ 32 nbytes = sendto(sock, &pmsg.data, strlen(pmsg.data), 0, (struct sockaddr*)&pmsg.addr, length);

- 파이프로부터 읽은 데이터를 에코

■ 37 ~ 58번 줄

- 부모 프로세스

■ 41 close(pipefd[0]);

- 읽기 파이프 닫음

■ 43 ~ 57번 줄

- 소켓으로부터 읽기


```

50     printf("부모 프로세스 : 소켓으로부터 %d 바이트를 받음\n", nbytes);
51     pmsg.data[nbytes]=0;
52
53     if(write(pipefd[1], (char *)&pmsg, sizeof(pmsg)) < 0)
54         perror("쓰기 실패");
55
56     printf("부모 프로세스 : 파이프에 쓰기\n", nbytes);
57 }
58 }
59
60 void errquit(char *mesg)
61 {
62     perror(mesg);
63     exit(1);
64 }
65
66 int main(int argc, char **argv)
67 {
68     struct sockaddr_in servaddr;
69     pid_t pid;
70     int sock, pipefd[2], port, length = sizeof(struct sockaddr);
71
72     if(argc!=2)
73     {
74         printf("\n Usage : %s port\n", argv[0]);
75         exit(EXIT_FAILURE);
76     }
77

```

■ 53 ~ 54번 줄

- 소켓으로부터 읽은 데이터를 파이프에 쓰기

```

78     port = atoi(argv[1]);
79     sock = socket(AF_INET, SOCK_DGRAM, 0);
80     if(sock < 0)
81     {
82         perror("소켓 생성 실패");
83         exit(EXIT_FAILURE);
84     }
85
86     bzero(&servaddr, length);
87     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
88     servaddr.sin_family = AF_INET;
89     servaddr.sin_port = ntohs(port);
90     bind(sock, (struct sockaddr*)&servaddr, length);
91
92     if(pipe(pipefd) == -1)
93         errquit("파이프 실패");
94     pid=fork();
95     if(pid < 0)
96         errquit("fork() 실패");
97     else if(pid>0)
98         parent_start(sock, pipefd);
99     else if(pid==0)
100         child_start(sock, pipefd);
101     return 0;
102 }

```

- 92 ~ 93번 줄
 - 파이프 생성

에코 클라이언트

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/socket.h>
8 #include <arpa/inet.h>
9 #include <netinet/in.h>
10
11 int main(int argc, char *argv[])
12 {
13     struct sockaddr_in peer;
14     int sock;
15     int nbytes;
16     char buf[512];
17
18     if(argc != 3)
19     {
20         printf("Usage : %d ip port\n", argv[0]);
21         exit(1);
22     }
23 }
```

```

24     sock = socket(AF_INET, SOCK_DGRAM, 0);
25     bzero(&peer, sizeof(struct sockaddr));
26     peer.sin_family = AF_INET;
27     peer.sin_port = htons(atoi(argv[2]));
28     peer.sin_addr.s_addr = inet_addr(argv[1]);
29
30     while(fgets(buf, sizeof(buf), stdin) != NULL)
31     {
32         nbytes = sendto(sock, buf, strlen(buf), 0, (struct sockaddr*)&peer, sizeof(peer));
33         if(nbytes < 0)
34         {
35             perror("보내기 실패");
36             exit(0);
37         }
38
39         nbytes = recvfrom(sock, buf, sizeof(buf)-1, 0, 0, 0);
40
41         if(nbytes < 0)
42         {
43             perror("받기 실패");
44             exit(0);
45         }
46
47         buf[nbytes] = 0;
48         fputs(buf, stdout);
49     }
50
51     close(sock);
52     exit(0);
53 }

```

■ 24 ~ 28번 줄

- UDP 소켓 생성 및 바인딩

■ 30 ~ 49번 줄

- 키보드 입력을 받고 서버로 보내는 부분

실행화면

서버 프로그램

```
$ pipe_echoserv 5555
대기중...
Parent : 13 bytes recv from socket
Parent : write to pipe
Child : read from pipe
Child : 13 bytes echo response
```

클라이언트 프로그램

```
$ udp_echocli 210.115.49.220 5555
test message
test message
```

실행

- 파일명 : pipe_echoserv.c
- 컴파일 : gcc -o pipe_echoserv pipe_echoserv.c
- 사용법 : pipe_echoserv 포트번호

- 파일명 : udp_echocli.c
- 컴파일 : gcc -o udp_echocli udp_echocli.c
- 사용법 : udp_echocli IP주소 포트번호

- 동작설명
 - 'test message'라는 빈칸포함 12바이트에 문장 끝의 'NULL'문자까지 13바이트가 쓰임
 - 부모 프로세스는 클라이언트로부터 받은 메시지를 파이프에 씀
 - 자식 프로세스는 파이프로부터 데이터를 읽어서 클라이언트로 에코 해줌

8.2 FIFO(First In First Out)

FIFO의 정의

임의의 프로세스에서 파이프에 접근하는 named pipe

8.2.1 FIFO 생성

mkfifo()

```
int mkfifo(const char *pathname, mode_t mode);
```

FIFO의 정의

- 파이프의 한계 극복
 - 파이프는 자식 · 부모 프로세스간의 통신에만 사용됨
 - 파이프에 이름을 지정해 가족관계가 아닌 임의의 다른 프로세스에서 접근

mkfifo()

- pathname 인자
 - 파이프의 이름 지정
 - 경로명 없이 파이프의 이름만 인자로 주면 현재 디렉토리에 FIFO 생성
- mode
 - 생성되는 FIFO의 파일 접근 권한 설정
 - 일반 파일 생성과 마찬가지로 최종적으로 umask 값에 영향을 받음

FIFO를 이용한 프로세스간 통신

FIFO를 사용하려면 `open()`해야함

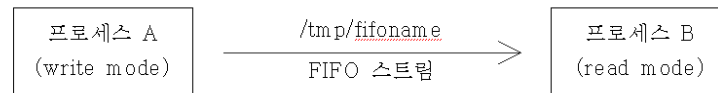


그림 8.3 FIFO를 이용한 프로세스간 통신

FIFO를 이용한 프로세스간 통신

- FIFO도 파이프의 일종으로 프로세스간에 스트림 채널을 형성
- FIFO는 파이프와 달리 파일 이름으로 액세스함

8.2.2 FIFO를 이용한 에코 서버 프로그램

에코 서버

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <sys/time.h>
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <errno.h>
12
13 #define MAX_BUFFER 512
14 #define FIFONAME "fifoname"
15
16 typedef struct message{
17     struct sockaddr_in addr;
18     char data[MAX_BUFFER];
19 }mesg_t;
20
```

- 17 struct sockaddr_in addr;
 - 클라이언트 주소
- 18 char data[MAX_BUFFER];
 - 읽은 데이터


```

21 void child_process(int sock)
22 {
23     mesg_t pmsg;
24     int nbytes, fford, length = sizeof(struct sockaddr);
25     fford = open(FIFONAME, O_RDONLY);
26
27     if(fford == -1)
28         errquit("fifo 열기 실패");
29
30     while(1)
31     {
32         nbytes = read(fford, (char *)&pmsg, sizeof(pmsg));
33         if(nbytes < 0)
34             errquit("읽기 실패");
35
36         printf("자식 프로세스 : FIFO로 부터 읽음\n", nbytes);
37         nbytes = sendto(sock, &pmsg.data, strlen(pmsg.data), 0, (struct sockaddr*)&pmsg.addr, length);
38         printf("자식 프로세스 : %d 바이트의 echo가 응답됨\n", nbytes);
39     }
40 }
41

```

■ 21 ~ 40번 줄

- 자식 프로세스 수행 부분

■ 25 fford = open(FIFONAME, O_RDONLY);

- 읽기 모드의 FIFO

■ 25 fford = open(FIFONAME, O_RDONLY);

- 읽기 모드로 FIFO open

■ 32 nbytes = read(fford, (char *)&pmsg, sizeof(pmsg));

- 파이프로부터 읽기 대기

■ 37 nbytes = sendto(sock, &pmsg.data, strlen(pmsg.data), 0, (struct sockaddr*)&pmsg.addr, length);

- 파이프로부터 읽은 데이터를 클라이언트로 전송

```

42 void parent_process(int sock)
43 {
44     msg_t pmsg;
45     int nbytes, fifowd, length=sizeof(struct sockaddr);
46
47     fifowd = open(FIFONAME, O_WRONLY);
48
49     if(fifowd == -1)
50         errquit("fifo 열기 실패");
51
52     printf("대기중...\n");
53
54     while(1)
55     {
56         nbytes = recvfrom(sock, (void*)&pmsg.data, MAX_BUFFER, 0, (struct sockaddr*)&pmsg.addr, &length);
57         if(nbytes < 0)
58             errquit("읽어오기 실패");
59         pmsg.data[nbytes] = 0;
60
61         printf("부모 프로세스 : 소켓으로부터 %d 바이트를 받음\n", nbytes);
62
63         if(write(fifowd, &pmsg, sizeof(pmsg)) < 0)
64             perror("쓰기 실패");
65
66         printf("부모 프로세스 : FIFO에 쓰기\n", nbytes);
67     }
68 }
69

```

■ 42 ~ 68번 줄

- 부모 프로세스 수행 부분

■ 47 `fifowd = open(FIFONAME, O_WRONLY);`

- 쓰기 모드의 FIFO

■ 47 `fifowd = open(FIFONAME, O_WRONLY);`

- 쓰기 모드로 FIFO open

■ 56 `nbytes = recvfrom(sock, (void*)&pmsg.data, MAX_BUFFER, 0, (struct sockaddr*)&pmsg.addr, &length);`

- 소켓으로부터 읽기 대기

■ 63 `if(write(fifowd, &pmsg, sizeof(pmsg)) < 0)`

- 소켓으로부터 읽은 데이터를 FIFO에 쓰기

```

70 void errquit(char *message)
71 {
72     perror(message);
73     exit(1);
74 }
75
76 int main(int argc, char **argv)
77 {
78     struct sockaddr_in servaddr;
79     pid_t pid;
80     int sock, port, length = sizeof(struct sockaddr);
81
82     if(argc != 2)
83     {
84         printf("Usage : %s port\n", argv[0]);
85         exit(EXIT_FAILURE);
86     }
87
88     port = atoi(argv[1]);
89     sock = socket(AF_INET, SOCK_DGRAM, 0);

```

- 89 sock = socket(AF_INET, SOCK_DGRAM, 0);
 - 소켓 생성

```

90     if(sock<0)
91     {
92         perror("소켓열기 실패");
93         exit(EXIT_FAILURE);
94     }
95
96     bzero(&servaddr, length);
97     servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
98     servaddr.sin_family = AF_INET;
99     servaddr.sin_port = ntohs(port);
100    bind(sock, (struct sockaddr *)&servaddr, length);
101
102    if(mkfifo(FIFONAME, 0660) == -1 && errno!=EEXIST)
103        errquit("FIFO생성 실패");
104
105    pid=fork();
106    if(pid < 0)
107        errquit("fork() 실패");
108    else if(pid > 0)
109        parent_process(sock);
110    else if(pid == 0)
111        child_process(sock);
112
113    return 0;
114 }

```

- 102 if(mkfifo(FIFONAME, 0660) == -1 && errno!=EEXIST)
 - FIFO 생성

실행화면

서버 프로그램

```
$ fifo_echoserv 5555
대기 중...
부모 프로세스 : 소켓으로부터 13 바이트를 받음
부모 프로세스 : FIFO에 쓰기
자식 프로세스 : FIFO로 부터 읽음
자식 프로세스 : 13 바이트의 echo가 응답됨
```

클라이언트 프로그램

```
$ udp_echocli 210.115.49.220 5555
test message
test message
```

fifo 파일 생성 확인

```
$ ls
fifo_echoserv* fifo_echoserv.c fifoname| pipe_echoserv* pipe_echoserv.c udp_echocli* udp_echocli.c
```

실행

- 파일명 : fifo_echoserv.c
- 컴파일 : gcc -o fifo_echoserv fifo_echoserv.c
- 사용법 : fifo_echoserv 포트번호
- 동작설명
 - 클라이언트 프로그램은 앞의 udp_echocli.c 프로그램 사용
 - 부모 프로세스는 클라이언트로부터 받은 메시지를 FIFO에 씀
 - 자식 프로세스는 FIFO로부터 데이터를 읽어서 클라이언트로 에코해줌
 - 디렉토리에 서버 프로그램에서 정한대로 FIFO가 생성됨
 - 여기서는 'fifoname' 으로 생성됨

8.3 메시지큐

8.3.1 메시지큐 개요

메시지큐의 정의

메시지 단위의 송수신용 메시지 큐

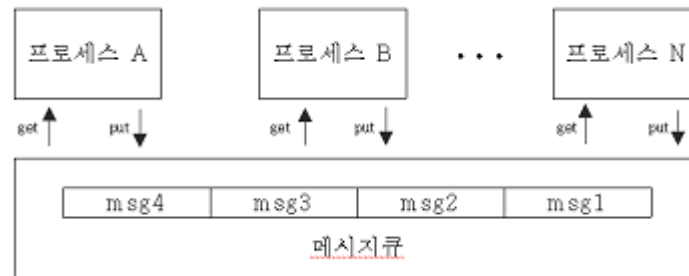


그림 8.4 메시지큐를 이용한 프로세스간 통신

메시지큐의 정의

- 특정 프로세스를 대상으로 메시지를 전송할 수 있음
- 메시지 전송에 우선순위를 줄 수 있음
- 메시지큐의 기능
 - 임의의 프로세스는 메시지큐에 메시지를 쓰거나(put), 읽을 수(get)있음
- 메시지큐는 일차원적인 큐를 제공하지 않음
 - 여러 개의 큐를 동시에 제공하는 2차원 큐로 제공됨

타입이 있는 메시지큐

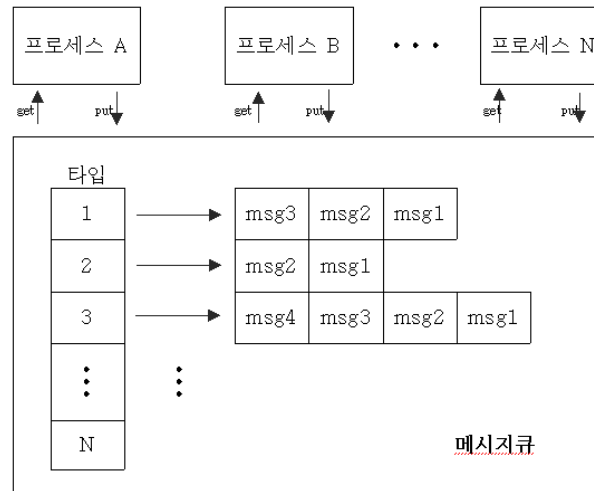


그림 8.5 타입이 있는 메시지큐

타입이 있는 메시지큐

■ 동작방식

- 각 타입의 메시지큐에 메시지가 있을 때
- 메시지 송수신 시 프로세스들은 타입을 지정하여야함
→ 메시지큐에 메시지를 쓰거나 읽을 때 타입을 지정
- 메시지를 쓸 때에 수신할 프로세스의 PID 값을 타입 값으로 지정
- 해당 프로세스에서는 자신의 PID를 타입 값으로 하여 메시지를 읽음
- 지정된 타입에서 꺼내올 메시지가 없을 경우
→ 그 보다 작은 타입의 큐에서 메시지를 꺼내옴

■ 메시지큐의 우선순위

- 타입 값을 우선순위로 사용하면 우선순위가 높은 타입 값이 큰 메시지를 먼저 수신
- 우선순위가 높은 메시지가 없는 때에만 우선순위가 낮은 메시지를 처리
- 메시지를 쓰인 순서나 우선순위 순으로 읽을 수 있음

8.3.2 메시지큐 생성

msgget()

```
int msgget(key_t key, int msgflg);
```

msgflg 설정

```
int mode = 0660;  
int msgflag = IPC_CREAT | mode;  
int msgid = msgget(key, size, msgflag);
```

msgflg 추가 설정

```
int mode = 0660;  
int msgflag = IPC_CREAT | IPC_EXCL | mode;  
int msgid = msgget(key, size, msgflag);
```

msgget()

- key
 - 메시지큐를 구분하기 위한 고유 키
 - 다른 프로세스에서 이 메시지큐에 접근하기 위해서는 key 값을 알아야함
- msgflg 인자
 - 메시지큐의 생성시 옵션을 지정
 - bitmask 형태의 인자를 취함
 - IPC_CREAT, IPC_EXCL등의 상수와 유닉스 파일 접근 권한도 bitmask 형태로 추가 지정 가능

msgflg 설정

- msgget()을 호출시 똑같은 key 값을 사용하는 메시지큐 존재 시 그 객체에 대한 ID를 리턴
- key 값에 대한 메시지큐 객체가 존재하지 않으면 새로운 메시지큐 객체를 생성하고 ID를 리턴

msgflg 추가 설정

- IPC_EXCL을 추가 설정하여 key 값을 사용하는 메시지큐 존재 시 msgget() 실패 후 -1 리턴
- IPC_EXCL은 IPC_CREAT와 같이 사용하여야 함
- key 값에 IPC_PRIVATE를 넣을 경우 key가 없는 메시지큐 생성
 - 메시지큐 ID를 공유하는 부모 · 자식 프로세스간은 통신하나 외부 프로세스는 접근 불가

메시지큐 객체 정의

```

1 struct msqid_ds{
2     struct ipc_perm msg_perm;
3     struct msg *msg_first;
4     struct msg *msg_last;
5     time_t msg_stime;
6     time_t msg_rtime;
7     time_t msg_ctime;
8     struct wait_queue *wwait;
9     struct wait_queue *rwait;
10    ushort msg_cbytes;
11    ushort msg_qnum;
12    ushort msg_qbytes;
13    ushort msg_lspid;
14    ushort msg_lrpid;
15 };
16
17 struct ipc_term{
18     key_t key;
19     ushort uid;
20     ushort gid;
21     ushort cuid;
22     ushort cgid;
23     ushort mode;
24     ushort seq;
25 };

```

메시지큐 객체 정의

- 2 struct ipc_perm msg_perm; - 메시지큐의 접근 권한
- 3 struct msg *msg_first; - 메시지큐의 처음 메시지
- 4 struct msg *msg_last; - 메시지큐의 마지막 메시지
- 5 time_t msg_stime; - 마지막으로 메시지가 송신된 시각
- 6 time_t msg_rtime; - 마지막으로 메시지가 수신된 시각
- 7 time_t msg_ctime; - 마지막으로 change가 수행된 시각
- 12 ushort msg_qbytes; - 메시지큐의 최대 바이트 수
- 13 ushort msg_lspid; - 마지막으로 msgsnd를 수행한 PID
- 14 ushort msg_lrpid; - 마지막으로 받은 PID
- 19 ushort uid; - owner의 euid
- 20 ushort gid; - owner의 egid
- 21 ushort cuid; - 생성자의 euid
- 22 ushort cgid; - 생성자의 egid
- 23 ushort mode; - 접근모드의 하위 9bit
- 24 ushort seq; - 순서번호

msgget() 사용 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/ipc.h>
5 #include <sys/msg.h>
6
7 int main(int argc, char **argv)
8 {
9     int msg_qid;
10    key_t key;
11
12    if(argc != 2)
13    {
14        printf("Usage : %s keyWn", argv[0]);
15        exit(0);
16    }
17
18    key = atoi(argv[1]);
19
20    if((msg_qid= msgget(key, IPC_CREAT | IPC_EXCL | 0666)) < 0)
21    {
22        perror("msgget() 실패");
23        exit(EXIT_FAILURE);
24    }
25
26    printf("생성된 메시지큐 ID = %dWn", msg_qid);
27
28    msg_qid = msgget(key, 0);
29    printf("key = %dWn", key);
30    printf("열린 메시지큐 ID = %dWn", msg_qid);
31
32    return 0;
33 }
```

msgget() 사용 예

- 20 ~ 24번 줄
 - 메시지큐 생성

- 26 printf("생성된 메시지큐 ID = %dWn", msg_qid);
 - 메시지큐의 ID 출력

- 28 msg_qid = msgget(key, 0);
 - key 값으로 이미 생성된 메시지큐 얻기

실행화면

```
$ msgget_test 7777  
생성된 메시지큐 ID = 98307  
key = 7777  
열린 메시지큐 ID = 98307
```

실행

- 파일명 : msgget_test.c
- 컴파일 : gcc -o msgget_test msgget_test.c
- 사용법 : msgget_test [key값으로 사용할 값]
- 동작설명
 - 메시지큐를 생성하고 메시지큐 ID를 출력

8.3.3 메시지 송수신

msgsnd()

메시지큐에 메시지를 넣는 함수

사용 문법

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

msgp 인자

```
struct msgbuf{  
    long mtype;  
    char mtext[1];  
};
```

msgsnd()

- msqid 인자
 - 메시지큐 객체의 ID
- msgp 인자
 - msgbuf 인자의 첫 4바이트는 반드시 long 타입이어야 함
 - mtype는 메시지 타입
 - 반드시 1 이상이어야 함
 - mtext
 - 메시지 데이터
 - 문자열일 필요는 없으며 binary등 임의의 데이터를 가리킬 수 있음
- msgsz 인자
 - 메시지 데이터의 길이를 나타내며 mtext의 크기만을 나타냄
- msgflg 인자
 - 0으로 한 경우
 - msgsnd() 호출시 메시지큐 공간이 부족하면 블록됨
 - IPC_NOWAIT로 한 경우
 - 메시지큐 공간이 부족한 경우 블록되지 않고 EAGAIN 에러코드와 -1을 리턴

msgrcv()

메시지큐로부터 메시지를 읽는 함수

사용문법

```
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);
```

msgrcv()

- msqid 인자
 - 메시지큐 객체의 ID

- msgp 인자
 - 메시지큐로부터 읽은 메시지를 저장하는 수신 공간
 - msgbuf 타입의 구조체를 가리킴

- msgsz 인자
 - 수신 공간의 크기

■ msgtype 인자

- 읽을 메시지의 타입을 지정
- 0이면 타입의 구분 없이 메시지큐에 입력된 순서대로 메시지를 읽음
- 음수 값을 넣으면 특별한 동작을 함
 - 예로 -10이면 타입이 10보다 같거나 작은 메시지를 읽는데 10인 메시지부터 우선순위
 - 우선순위 큐의 구현에 사용할 수 있음

■ msgflag 인자

- 메시지큐에 메시지가 없는 경우 취할 동작을 지정
- 0이면 데이터가 없을 때 msgrcv() 함수는 대기
- IPC_NOWAIT이면 EAGAIN 에러코드와 -1 리턴
- 읽을 메시지가 수신 공간의 크기보다 클 경우
 - E2BIG 에러 발생
- MSG_NOERROR로 설정 시 msgsz 크기만큼만 읽고 뒷부분은 잘려짐

메시지큐 이용 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/ipc.h>
7
8 #define BUFSZ 512
9
10 typedef struct _msg{
11     long msg_type;
12     char msg_text[BUFSZ];
13 }msg_t;
14
15 int main(int argc, char **argv)
16 {
17     pid_t pid;
18     int length, qid;
19     msg_t pmsg;
20     key_t key;
21 }
```

- 10 ~ 13번 줄
 - 메시지 내용 구조체 선언

- 19 msg_t pmsg;
 - 전송할 메시지

- 20 key_t key;
 - 메시지큐 key

```

22     if(argc != 2)
23     {
24         printf("Usage : %s msqkey", argv[0]);
25         exit(EXIT_FAILURE);
26     }
27
28     key = atoi(argv[1]);
29
30     if((qid = msgget(key, IPC_CREAT | 0600)) < 0)
31     {
32         perror("msgget");
33         exit(EXIT_FAILURE);
34     }
35
36     if((pid=fork()) < 0)
37     {
38         printf("fork() 실패");
39         exit(EXIT_FAILURE);
40     }
41     else if(pid > 0)
42     {
43         int nbytes;
44         msg_t rmsg;

```

■ 30 ~ 34번 줄

- 메시지큐 생성

■ 41 ~ 53번 줄

- 부모 프로세스 작업


```

45     printf("Wn%d 프로세스 메시지큐 읽기 대기중..Wn", getpid());
46     nbytes = msgrcv(qid, &rmsg, BUFSZ, getpid(), 0);
47     printf("recv = %d 바이트 Wn", nbytes);
48     printf("type = %ld Wn", rmsg.msg_type);
49     printf("수신 프로세스 PID = %dWn", getpid());
50     printf("value = %s Wn", rmsg.msg_text);
51     msgctl(qid, IPC_RMID, 0);
52     exit(EXIT_SUCCESS);
53 }
54
55 if(fgets((&pmsg)->msg_text, BUFSZ, stdin) == NULL)
56 {
57     puts("post에 데이터가 없습니다.");
58 }
59
60 pmsg.msg_type = getppid();
61 length=strlen(pmsg.msg_text);
62
63 if((msgsnd(qid,&pmsg,length,0))<0)
64 {
65     perror("msgsnd 실패");
66     exit(EXIT_FAILURE);
67 }
68
69 return 0;
70 }

```

- 51 msgctl(qid, IPC_RMID, 0);
 - 메시지큐 삭제

- 55 ~ 68번 줄
 - 자식 프로세스 작업

- 60 pmsg.msg_type = getppid();
 - 부모 프로세스의 ID를 메시지 타입으로 사용

- 63 ~ 67번 줄
 - 메시지 전송

실행화면

```
$ msgsend_pid 5555
5885 프로세스 메시지큐 읽기 대기중..
test message
recv = 13 바이트
type = 5885
수신 프로세스 PID = 5885
value = test message
```

실행

- 파일명 : msgsend_pid.c
- 컴파일 : gcc -o msgsend_pid msgsend_pid.c
- 사용법 : msgsend_pid [key값으로 사용할 값]
- 동작설명
 - msgtype 값을 특정 프로세스의 PID로 지정하여 메시지 전달
 - 지정된 프로세스에서 메시지를 수신

8.3.4 메시지큐 제어

msgctl()

사용 문법

```
int msgctl(int msqid, int 증, struct msqid_ds *buf)
```

msgctl()

- 메시지큐를 제어
 - 정보 읽기, 동작 허가 권한 변경, 메시지큐 삭제 등
- 사용 문법
 - msqid
 - 메시지큐 객체 ID
 - cmd
 - 제어 명령 구분
 - IPC_STAT로 메시지큐 객체를 얻은 후 메시지큐 객체를 변경 후 IPC_SET호출
 - IPC_STAT : 메시지큐 객체에 대한 정보를 얻어오는 명령
 - IPC_SET : r/w 권한, euid, egid, msg_qbytes를 변경하는 명령
 - IPC_RMID : 메시지큐를 삭제하는 명령

msgctl()사용 예

```
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/msg.h>
7 #include <unistd.h>
8
9 int msq_remove(int qid)
10 {
11     if((msgctl(qid,IPC_RMID,NULL)) < 0)
12     {
13         perror("msgctl");
14         return -1;
15     }
16     printf("%d메시지큐 삭제됨\n", qid);
17     return 0;
18 }
19
20 int view_qinfo(int qid)
21 {
22     struct msqid_ds buf;
23     struct ipc_perm *pm;
24
25     if((msgctl(qid, IPC_STAT, &buf)) < 0)
26     {
27         perror("msgctl");
28         return -1;
29     }
30 }
```

- 9 ~ 18번 줄
 - 메시지큐 제거

- 20 ~ 37번 줄
 - 메시지큐 정보 출력

- 25 ~ 29번 줄
 - 메시지큐 객체 얻음

```

31     pm=&buf.msg_perm;
32     printf("큐의 최대 바이트수 :%d Wn", buf.msg_qbytes);
33     printf("큐의 유효 사용자 UID : %dWn", pm->uid);
34     printf("큐의 유효 사용자 GID : %dWn", pm->gid);
35     printf("큐 접근 권한 : 0%oWn", pm->mode);
36     return 0;
37 }
38
39 int main(int argc, char **argv)
40 {
41     int qid;
42     key_t key;
43
44     if(argc != 2)
45     {
46         printf("Usage : %s msqkeyWn", argv[0]);
47         exit(EXIT_FAILURE);
48     }
49     key=atoi(argv[1]);
50
51     if((qid=msgget(key, 0)) < 0)
52     {
53         perror("msgget 실패");
54         exit(0);
55     }
56
57     view_qinfo(qid);
58     msq_remove(qid);
59     exit(EXIT_SUCCESS);
60 }

```

- 57 view_qinfo(qid);
 - 메시지큐 정보 출력 함수 호출

- 58 msq_remove(qid);
 - 메시지큐 제거 함수 호출

실행 화면

```
$ msgctl 7777
큐의 최대 바이트수 :16384
큐의 유효 사용자 UID : 517
큐의 유효 사용자 GID : 517
큐 접근 권한 : 0666
98307메시지큐 삭제됨
```

실행

- 파일명 : msgctl.c
- 컴파일 : gcc -o msgctl msgctl.c
- 사용법 : msgsend_test [key값으로 사용할 값]
- 동작설명
 - IPC_STAT 명령으로 메시지큐의 객체 얻어와서 정보 출력
 - IPC_RMID 명령으로 메시지큐 삭제

8.3.5 메시지큐를 이용한 에코 서버

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <netinet/in.h>
7 #include <errno.h>
8 #include <sys/ipc.h>
9 #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <sys/msg.h>
12
13 #define MAX_BUF 512
14 #define RES_SEND_PROC "res_send_proc"
15
16 typedef struct _message{
17     long message_type;
18     struct sockaddr_in addr;
19     char message_text[MAX_BUF];
20 }msg_t;
21
```

- 16 ~ 20번 줄
 - 메시지 구조 정의

```

22 void fork_and_exec(char *key, char *port)
23 {
24     pid_t pid = fork();
25     if(pid < 0)
26         errquit("fork() 실패");
27     else if(pid>0)
28         return;
29
30     execlp(RES_SEND_PROC, RES_SEND_PROC, key, port, 0);
31     perror("execlp() 실패");
32 }
33
34 int errquit(char *msg)
35 {
36     perror(msg);
37     exit(1);
38     return 0;
39 }
40
41 int main(int argc, char *argv[])
42 {
43     struct sockaddr_in servaddr;
44     msg_t pmsg;
45     key_t key;
46     int msqid, size, nbytes, sock, port, length = sizeof(struct sockaddr);
47

```

■ 22 ~ 32번 줄

- 에코 서버 프로그램이 시작되는 RES_SEND_PROC 호출


```

48     if(argc != 3)
49     {
50         printf("Usage : %s msgq_key port\n", argv[0]);
51         exit(1);
52     }
53
54     key = atoi(argv[1]);
55     port = atoi(argv[2]);
56
57     msqid = msgget(key, IPC_CREAT | 0600);
58     if(msqid == -1)
59         errquit("msgget 실패");
60
61     sock = socket(AF_INET, SOCK_DGRAM, 0);
62
63     if(sock < 0)
64         errquit("socket 실패");
65
66     bzero(&servaddr, length);
67     servaddr.sin_port = htons(port);
68     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
69     servaddr.sin_family = AF_INET;
70
71     if(bind(sock, (struct sockaddr*)&servaddr, length) < 0)
72         errquit("bind() 실패");
73

```

- 57 `msqid = msgget(key, IPC_CREAT | 0600);`
 - 메시지큐 생성

- 61 ~ 72번 줄
 - 소켓 생성 및 바인딩

```

74     fork_and_exec(argv[1], argv[2]);
75     fork_and_exec(argv[1], argv[2]);
76     fork_and_exec(argv[1], argv[2]);
77
78     pmsg.message_type = 1;
79     size = sizeof(msg_t) - sizeof(long);
80     puts("서버 대기 시작");
81
82     while(1)
83     {
84         nbytes = recvfrom(sock, pmsg.message_text, MAX_BUF, 0,
                           (struct sockaddr *)&pmsg.addr, &length);
85
86         if(nbytes < 0)
87         {
88             perror("받기 실패");
89             continue;
90         }
91         pmsg.message_text[nbytes] = 0;
92
93         if(msgsnd(msqid, &pmsg, size, 0) == -1)
94             errquit("msgsnd() 실패");
95     }
96     return 0;
97 }

```

- 92 ~ 93번 줄
 - 메시지큐에 쓰기

RES_SEND_PROC 구현

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <netinet/in.h>
6 #include <fcntl.h>
7 #include <errno.h>
8 #include <signal.h>
9 #include <sys/ipc.h>
10 #include <sys/socket.h>
11 #include <sys/types.h>
12 #include <sys/msg.h>
13 #include <sys/stat.h>
14
15 #define MAX_BUF 512
16
17 typedef struct _message{
18     long message_type;
19     struct sockaddr_in addr;
20     char message_text[MAX_BUF];
21 }msg_t;
22
```

- 17 ~ 21번 줄
 - 메시지 구조 정의

```

23 int qread_and_echoreply(int msqid, int sock)
24 {
25     int size, length = sizeof(struct sockaddr);
26     msg_t pmsg;
27     size = sizeof(msg_t) - sizeof(long);
28     pmsg.message_type = 0;
29     while(1)
30     {
31         if(msgrcv(msqid, (void *)&pmsg, size, 0, 0) < 0)
32             errquit("메시지 받기 실패");
33         printf("응답된 프로세스 PID = %d\n", getpid());
34         printf(pmsg.message_text);
35         if(sendto(sock, pmsg.message_text, strlen(pmsg.message_text), 0,
36                 (struct sockaddr*)&pmsg.addr, length) < 0)
37             errquit("서버로 보내기 실패");
38         pmsg.message_text[0] = 0;
39     }
40     return 0;
41 }
42 int errquit(char *msg)
43 {
44     perror(msg);
45     exit(1);
46     return 0;
47 }
48

```

■ 23 ~ 40번 줄

- 메시지큐에서 읽어서 응답하기

```

49 int main(int argc, char **argv)
50 {
51     struct sockaddr_in servaddr;
52     key_t key;
53     int sock, port, msqid, length = sizeof(struct sockaddr);
54
55     key = atoi(argv[1]);
56     port = atoi(argv[2]);
57     if((msqid=msgget(key, 0)) == -1)
58         errquit("rep msgget 실패");
59
60     sock = socket(AF_INET, SOCK_DGRAM, 0);
61     bzero(&servaddr, length);
62     servaddr.sin_family=AF_INET;
63     servaddr.sin_port = htons(port);
64     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
65     bind(sock, (struct sockaddr*)&servaddr, length);
66
67     qread_and_echoreply(msqid, sock);
68
69     return 0;
70 }

```

■ 60 ~ 65번 줄

- 소켓 생성 및 바인딩

■ 67 qread_and_echoreply(msqid, sock);

- msg queue에서 읽어서 reply

실행화면

서버 실행 화면

```
$ echoserver_msgq 1234 5555
서버 대기 시작
응답된 프로세스 PID = 8397
test message
응답된 프로세스 PID = 8398
test message2
응답된 프로세스 PID = 8399
test message3
```

클라이언트 실행 화면

```
$ udp_echocli 210.115.49.220 5555
test message
test message
test message2
test message2
test message3
test message3
```

실행

- 파일명 : echoserver_msgq.c
- 컴파일 : gcc -o echoserver_msgq echoserver_msgq.c
- 사용법 : echoserver_msgq [key값으로 사용할 값] 포트번호

- 파일명 : res_send_proc.c
- 컴파일 : gcc -o res_send_proc res_send_proc.c
- 사용법 : echoserver_msgq에서 execlp()로 실행됨

- 파일명 : udp_echocli.c
- 컴파일 : gcc -o udp_echocli udp_echocli.c
- 사용법 : udp_echocli IP주소 포트번호

- 동작설명
 - res_send_proc.c 는 exec가 호출될 때 인자로 넘어온 msgq key와 port 값으로부터 해당 메시지큐와 키와 포트번호를 얻음
 - 메시지큐에서 읽기를 대기하다가 메시지가 도착하면 클라이언트에게 응답 메시지 전송
 - 여러 프로세스들이 순서대로 에코를 처리함
 - PID 8397, 8398, 8399를 통해 확인

8.4 공유메모리

공유메모리의 정의

프로세스들이 공통으로 사용할 수 있는 메모리 영역

공유메모리의 정의

- 프로세스들이 공통으로 사용하는 메모리 영역
 - 어떤 프로세스에서 사용 중인 메모리는 그 프로세스만 접근가능
 - 공유메모리는 특정 메모리 영역을 다른 프로세스와 공유
 - 프로세스간 통신 가능
 - 통신에서 데이터를 한 번 읽어도 데이터가 계속 남아 있음
 - 같은 데이터를 여러 프로세스가 중복하여 읽을 경우 효과적

8.4.1 공유메모리 생성

shmget()

```
int shmget(key_t key, int, size, int msgflg);
```

shmflg 설정

```
int mode = 0660;  
int shmflag = IPC_CREAT | mode;  
int shmid = shmget(key, size, shmflag);
```

shmflg 추가 설정

```
int mode = 0660;  
int shmflag = IPC_CREAT | IPC_EXCL | mode;  
int shmid = shmget(key, size, shmflag);
```

shmget()

- key
 - 공유메모리를 구분하기 위한 고유 키
 - 다른 프로세스에서 이 공유메모리에 접근하기 위해서는 key 값을 알아야함
- shmflg 인자
 - 공유메모리의 생성시 옵션을 지정
 - bitmask 형태의 인자를 취함
 - IPC_CREAT, IPC_EXCL등의 상수와 유닉스 파일 접근 권한도 bitmask 형태로 추가 지정 가능

shmflg 설정

- shmget()을 호출시 똑같은 key 값을 사용하는 공유메모리 존재 시 그 객체에 대한 ID를 리턴
- key 값에 대한 공유메모리 객체가 존재하지 않으면 새로운 공유메모리 객체를 생성하고 ID를 리턴

shmflg 추가 설정

- IPC_EXCL을 추가 설정하여 key 값을 사용하는 공유메모리 존재 시 shmget() 실패 후 -1 리턴
- IPC_EXCL은 IPC_CREAT와 같이 사용하여야 함
- key 값에 IPC_PRIVATE를 넣을 경우 key가 없는 공유메모리 생성
- 공유메모리 ID를 공유하는 부모 · 자식 프로세스간은 통신하나 외부 프로세스는 접근 불가

공유메모리 객체구조

```
1 struct shmid_ds{
2     struct ipc_perm shm_perm;
3     int shm_segsz;
4     time_t shm_atime;
5     time_t shm_dtime;
6     time_t shm_ctime;
7     unsigned short shm_cpid;
8     unsigned short shm_lpid;
9     short shm_nattch;
10
11     unsigned short shm_npages;
12     unsigned long *shm_pages;
13     struct vm_area_struct *attaches;
14 }
```

- 2 struct ipc_perm shm_perm; - 동작 허가 사항
- 3 int shm_segsz; - 세그먼트의 크기
- 4 time_t shm_atime; - 마지막 attach 시각
- 5 time_t shm_dtime; - 마지막 detach 시각
- 6 time_t shm_ctime; - 마지막 change 시각
- 7 unsigned short shm_cpid; - 생성자의 PID
- 8 unsigned short shm_lpid; - 마지막 접근자의 PID
- 9 short shm_nattch; - 현재 attaches no.

- 11 ~ 13번 줄
 - private 값임

- 11 unsigned short shm_npages; - 세그먼트의 크기
- 12 unsigned long *shm_pages; - array of ptrs to frames -> SHMMAX
- 13 struct vm_area_struct *attaches; - descriptors for attaches

공유메모리 첨부

공유메모리의 물리적 주소를 자신의 프로세스의 가상메모리 주소로 매핑

shmat()

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

공유메모리 첨부

- 사용전에 매핑 하는 것
- shmat()
 - shmid인자 : 공유메모리 객체 ID
 - shmaddr : 프로세스의 메모리 주소 중 첨부시킬 주소
 - 0을 넣으면 커널이 자동으로 빈 공간을 찾아서 처리함
 - shmflg
 - SHM_RDONLY : 공유메모리를 읽기 전용으로 첨부
 - 0 : 공유메모리에 읽기, 쓰기 가능
- 정상 작동시 프로세스 내의 첨부된 주소 리턴
- 에러 발생시 NULL 포인터 리턴

공유메모리의 분리

공유메모리 사용 종료 후 자신이 사용하던 메모리 영역에서 분리

shmdt()

```
int shmdt(const void *shmaddr);
```

공유메모리의 분리

- shmdt()
 - shmaddr 인자
 - shmdt()가 리턴했던 주소
 - 현재 프로세스에 첨부된 공유메모리의 시작주소
- 공유메모리가 메모리에서 삭제된것이 아님
 - 현재 프로세스에서만 사용하지 못함
 - 다른 프로세스에서는 계속 사용 가능
- 공유메모리 객체 shmid_ds 구조체의 shm_nattch 멤버 변수
 - 현재 공유메모리를 첨부하고 있는 프로세스의 수를 나타냄
 - shmat() 호출시 1증가
 - shmdt() 호출시 1감소

8.4.2 공유메모리 제어

shmctl()

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmctl()

- 공유메모리를 제어
 - 정보 읽기, 동작 허가 권한 변경, 공유메모리 삭제 등
- 사용 문법
 - shmid
 - 공유메모리 객체 ID
 - cmd
 - 제어 명령 구분
 - IPC_STAT : 공유메모리 객체에 대한 정보를 얻어오는 명령
 - struct shmid_ds *buf 인자에 공유메모리 객체 복사
 - IPC_SET : r/w 권한, euid, egid, msg_qbytes를 변경하는 명령
 - IPC_RMID : 공유메모리를 삭제하는 명령
 - buf
 - cmd에 따라 의미가 바뀌는 인자
 - 공유메모리 객체 정보를 얻어오는 명령어에서는 얻어온 객체를 buf에 저장
 - 동작 허가 권한을 변경하는 명령어에서는 변경할 내용을 buf에 저장

공유메모리 정보 얻기

공유메모리 객체 정보 얻기

```
struct shmid_ds shmds;  
shmctl(shmid, IPC_STAT, buf);
```

공유메모리 객체의 동작허가 권한 변경

```
shmctl(shmid, IPC_SET, &shmds);
```

공유메모리 객체의 접근 권한 변경

```
1 struct shmid_ds shmds;  
2 shmctl(shmid, IPC_STAT, &shmds);  
3 shmds.shm_perm.uid = changeuid;  
4 shmds.shm_perm.gid = changegid;  
5 shmds.shm_perm.mode = changemode;  
6 shmctl(shmid, IPC_SET, &shmds);
```

공유메모리 정보 얻기

■ 공유메모리 객체 정보 얻기

- IPC_STAT 명령으로 공유메모리 객체 정보 얻음
- buf 인자에 저장

■ 공유메모리 객체의 동작허가 권한 변경

- 기존의 공유메모리 객체의 shmds.shm_perm.uid, shmds.shm_perm.gid, shmds.shm_perm.mode 값만 변경됨
- 위의 값들이 변경되면 shmid_ds의 chm_ctime 값이 변경된 시각으로 바뀜

■ 공유메모리 객체의 접근 권한 변경

- 1 struct shmid_ds shmds; : 공유메모리 객체 선언
- 2 shmctl(shmid, IPC_STAT, &shmds); : 공유메모리 객체 얻어옴
- 3 shmds.shm_perm.uid = changeuid; : 실행 권한 uid 값을 변경
- 4 shmds.shm_perm.gid = changegid; : 실행 권한 gid 값을 변경
- 5 shmds.shm_perm.mode = changemode; : 접근 권한 변경
- 6 shmctl(shmid, IPC_SET, &shmds); : 공유메모리 객체 내용 변경

공유메모리 삭제

shmctl()의 IPC_RMID 명령을 이용

```
shmctl(shmid, IPC_RMID, 0);
```

공유메모리 삭제

- 공유메모리 삭제 요청시 하나 이상의 다른 프로세스가 이 공유메모리를 사용시
 - 공유메모리는 삭제되지 않음
 - shm_nattach 값이 0이 될 때까지 기다린 후 삭제됨

8.4.3 공유메모리의 동기화문제 처리

동기화문제

하나의 공유데이터를 둘 이상의 프로세스가 동시에 접근하는 문제

동기화문제 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <sys/ipc.h>
7 #include <sys/shm.h>
8 #include <sys/types.h>
9
10 char *shm_data;
11 int shmid;
12
13 void errquit(char *message)
14 {
15     perror(message);
16     exit(1);
17 }
18
```

- 10 char *shm_data;
 - 공유메모리 포인터

- 11 int shmid;
 - 공유메모리 ID

```

19 void fork_and_run()
20 {
21     pid_t pid = fork();
22     if(pid < 0)
23         errquit("fork() 실패");
24     else if(pid == 0)
25     {
26         busy();
27         exit(0);
28     }
29     return;
30 }
31
32 int busy()
33 {
34     int i = 0;
35     for(i=0 ; i<50000 ; i++)
36         access_shm(i);
37     shmdt(shm_data);
38
39     return 0;
40 }
41

```

■ 19 ~ 30번 줄

- 자식 프로세스를 생성하여 busy() 함수를 호출하도록 함

■ 32 ~ 40번 줄

- 각 프로세스가 경쟁적으로 공유메모리에 접근

■ 37 shmdt(shm_data);

- 공유메모리 분리


```

42 int access_shm(int count)
43 {
44     int i;
45     pid_t pid;
46
47     sprintf(shm_data, "%d", getpid());
48
49     for(i=0 ; i<1000 ; i+ );
50     pid = atoi(shm_data);
51
52     if(pid != getpid())
53         printf("Error(count=%d) : 동기화문제 발생 \n", count);
54     else
55     {
56     }
57
58     return 0;
59 }
60
61 int main(int argc, char *argv[])
62 {
63     key_t shmkey;
64

```

- 42 ~ 59번 줄
 - 공유메모리에 접근하는 함수
- 47 `sprintf(shm_data, "%d", getpid());`
 - 공유메모리에 자신의 PID 기록
- 49 `for(i=0 ; i<1000 ; i++);`
 - 공유메모리 접근 시간에 포함
- 52 ~ 56번 줄
 - 공유메모리에 기록한 PID가 자신의 PID가 아니면 error 출력
 - 정상이면 아무 동작 없음
- 63 `key_t shmkey;`
 - 공유메모리 키

```

66     if(argc<2)
67     {
68         printf("Usage : %s shmkeyWn", argv[0]);
69         exit(1);
70     }
71     shmkey = atoi(argv[1]);
72     shmid = shmget(shmkey, 128, IPC_CREAT | 0600);
73
74     if(shmid < 0)
75         errquit("shmget 실패");
76
77     shm_data=(char *)shmat(shmid, (void*)0, 0);
78     if(shm_data == (char *) - 1)
79         errquit("shmat 실패");
80
81     fork_and_run();
82     fork_and_run();
83
84     busy();
85     wait(NULL);
86     shmctl(shmid, IPC_RMID, 0);
87
88     return 0;
89 }

```

- 86 shmctl(shmid, IPC_RMID, 0);
 - 공유메모리 제거

실행 화면

```
$ shm_access 5555
Error(count=142) : 동기화문제 발생
Error(count=7) : 동기화문제 발생
Error(count=150) : 동기화문제 발생
Error(count=9) : 동기화문제 발생
Error(count=152) : 동기화문제 발생
Error(count=11) : 동기화문제 발생
Error(count=154) : 동기화문제 발생
Error(count=13) : 동기화문제 발생
Error(count=156) : 동기화문제 발생
Error(count=15) : 동기화문제 발생
Error(count=158) : 동기화문제 발생
Error(count=17) : 동기화문제 발생
Error(count=160) : 동기화문제 발생
Error(count=19) : 동기화문제 발생
```

실행

- 파일명 : shm_access.c
- 컴파일 : gcc -o shm_access shm_access.c
- 사용법 : shm_access [key값으로 사용할 값]
- 동작설명
 - 공유메모리를 생성하고 fork()를 두 번 호출하여 자식 프로세스를 두 개 생성함
 - 부모 프로세스를 포함한 3개의 프로세스가 busy()를 통해 공유메모리에 경쟁적으로 접근함

8.5 세마포어

8.5.1 세마포어 사용

세마포어

한 순간에 공유데이터를 액세스하는 프로세스 수를 하나로 제한

세마포어 생성

`semget()`

```
int semget(key_t key, int nsems, int semflg);
```

세마포어

- 세마포어의 정의
 - 어떤 공유데이터에 대해 현재 사용 가능한 데이터의 수
- 크리티컬 영역
 - 동기화문제가 발생할 수 있는 코드 블록

세마포어 생성

- `semget()`
 - `key` : 세마포어를 구분하기 위해 지정해 주는 키
 - `nsems` : 세마포어 집합을 구성하는 멤버의 수
 - `semflg` : 세마포어 생성에 관한 옵션을 설정하거나 세마포어 객체에 대한 접근 권한 설정
- `semget()`
 - 성공적으로 수행시 세마포어 집합에 관한 정보를 담고 있는 세마포어 객체 생성
 - 세마포어 객체 ID를 리턴

플래그 사용 예

IPC_CREAT

```
semget(key, nsems, IPC_CREAT | mode);
```

IPC_EXCL

```
mode = 0660;  
semget(key, nsems, IPC_CREAT | IPC_EXCL | mode);
```

플래그 사용 예

■ IPC_CREAT

- key에 해당하는 세마포어 객체가 존재하면 기존의 세마포어 ID를 리턴
- key에 해당하는 세마포어 객체가 없으면 새로 생성하고 세마포어 객체 ID 리턴
- `semget(key, 0, 0)`
 - key값을 갖는 세마포어의 ID를 단순히 알기위해 사용

■ IPC_EXCL

- key값에 `IPC_PRIVATE`를 사용하면 key가 없는 세마포어 객체 생성
 - 키가 없는 세마포어 이므로 다른 프로세스에서 접근 못함
- 세마포어 생성 후 `fork()`를 호출하면 자식 프로세스는 세마포어 객체의 ID를 상속받음
 - `semget()` 함수를 사용하지 않고 세마포어에 접근

세마포어 객체

```
1 struct semid_ds{
2     struct ipc_perm sem_perm;
3     time_t sem_otime;
4     time_t sem_ctime;
5     struct sem *sem_base;
6     struct wait_queue *eventn;
7     struct wait_queue *eventz;
8     struct sem_undo *undo;
9     ushort sem_nsems;
10 };
```

세마포어 객체

- 2 struct ipc_perm sem_perm;
 - 접근 허가 내용
- 3 time_t sem_otime;
 - 최근 세마포어 조작 시간
- 4 time_t sem_ctime;
 - 최근 변경 시각
- 5 struct sem *sem_base;
 - 첫 세마포어 포인터
- 9 ushort sem_nsems;
 - 세마포어 멤버 수

세마포어 연산

세마포어의 값을 증가 또는 감소하는 것

semop()

```
int semop(int semid, struct sembuf *operations, unsigned nsops);
```

sembuf 타입 구조체

```
1 struct sembuf{  
2     short sem_num;  
3     short sem_op;  
4     short sem_flg;  
5 }
```

세마포어 연산

■ semop()

- semid : 세마포어 ID
- operations : sembuf 타입의 구조체를 가짐
 - 2 short sem_num; - 멤버 세마포어 번호(첫 번째 멤버 세마포어는 0)
 - 3 short sem_op; - 세마포어 연산 내용
 - 4 short sem_flg; - 조작 플래그, 0, IPC_NOWAIT, SEM_UNDO등의 값을 bitmask 형태로 가질수 있음
 - IPC_NOWAIT시 세마포어 값이 부족해도 블록되지 않음
 - SEM_UNDO시 프로세스의 종료 시 커널은 해당 세마포어 연산 취소
- nsops : 두 번째 인자 operations 구조체가 몇 개의 리스트를 가지고 있는지 나타냄

8.5.2 세마포어 제어

semctl()

```
int semctl(int semid, int member_index, int cmd, union semun semarg);
```

semctl()

- 세마포어의 사용종료, 세마포어 값 읽기 및 설정, 특정 멤버 세마포어를 기다리는 프로세스의 수 알기에 사용
- semid
 - 제어할 대상의 세마포어 ID
- member_index
 - 세마포어 멤버 번호
- cmd
 - 수행할 동작
- semun 타입의 semarg 인자
 - cmd의 종류에 따라 각각 다르게 사용될 수 있는 인자

semum 공용체

```
1 union semum{
2     int val;
3     struct semid_ds *buf;
4     unsigned short int *array;
5     struct seminfo *__buf;
6     void *__pad;
7 };
```

semun 공용체

- 2 int val;
 - SETVAL을 위한 값
- 3 struct semid_ds *buf;
 - IPC_STAT, IPC_SET을 위한 버퍼
- 4 unsigned short int *array;
 - GETALL, SETALL을 위한 배열
- 5 struct seminfo *__buf;
 - IPC_INFO를 위한 버퍼
- 6 void *__pad;
 - dummy

IPC_STAT

```
struct semid_ds semobj;  
union semun semarg;  
semarg.buf = &semobj;  
semctl(semid, 0, IPC_STAT, semarg);
```

SETVAL

```
union semun semarg;  
unsigned short semvalue = 5;  
semarg.val = semvalue;  
semctl(semid, 1, SETVAL, semarg);
```

IPC_STAT

- semid_ds타입의 세마포어 객체를 얻어오는 명령
- union semun semarg인자에 세마포어 객체가 리턴
- semarg.buf 포인터를 통해 세마포어 객체를 접근
- IPC_STAT 명령을 사용할 때엔 semctl()하수의 member_index 인자는 사용되지 않음(0으로 설정)

SETVAL

- 세마포어를 생성한 후에 공유데이터의 수를 설정해 주는 작업
- 세마포어 객체의 초기화를 하는 함수
- union semun semarg 인자에는 설정하려는 세마포어의 값을 지정
- semarg인자의 semarg.val 변수에 원하는 초기 값을 입력
- 위의 예에서는 1번 멤버 세마포어의 값을 5으로 설정

SETALL

```
unsigned short values = {1, 2, 3, 4};  
union semun semarg;  
semarg.array = values;  
semctl(semid, 0, SETALL, semarg);
```

GETVAL

```
int n semctl(semid, 1, GETVAL, 0);
```

GETALL

```
union semun semarg;  
unsigned short semvalues[4];  
semarg.array = semvalues;  
semctl(semid, 0, GETALL, semarg);
```

SETALL

- 세마포어 집합 내의 모든 세마포어의 값을 초기화하는 명령
- union semun semarg 인자의 semarg.array인자에 초기 값을 배열 형태로 넣어줌
- member_index는 사용되지 않음
- 위의 예제에서는 세마포어 집합의 멤버 세마포어 수는 4이며 1, 2, 3, 4로 초기화

GETVAL

- 특정 멤버 세마포어의 현재 값을 얻어오는 명령
- 위의 예제에서는 1번 멤버 세마포어 값을 얻어옴

GETALL

- 모든 멤버 세마포어의 현재 값을 읽음
- 위의 예제에서는 세마포어 집합의 멤버 수가 4인 경우 각 멤버 세마포어 값을 읽음

GETNCNT

```
semctl(semid, member_index, GETNCNT, 0);
```

GETPID

```
int pid = semctl(semid, member_index, GETPID, 0);
```

IPC_RMID

```
semctl(semid, 0, IPC_RMID, 0);
```

GETNCNT

- 세마포어 값이 원하는 값 이상으로 증가되기를 기다리는 프로세스의 수를 얻음
 - 특정 멤버 세마포어를 사용하기 위해 블록되어 있는 프로세스의 수
- semctl() 함수의 리턴 값으로 세마포어를 기다리는 프로세스의 수를 얻음
- union semun semarg 인자는 사용되지 않음

GETPID

- 특정 멤버 세마포어에 대해 마지막으로 semop() 함수를 수행한 프로세스 PID를 얻음

IPC_RMID

- 세마포어를 삭제

8.5.3 세마포어 이용 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <fcntl.h>
6 #include <sys/ipc.h>
7 #include <sys/sem.h>
8 #include <sys/types.h>
9
10 #define ok 0
11 #define no 1
12
13 int semid;
14
15 struct sembuf increase[] = { { 0, +1, SEM_UNDO}, {1, +1, SEM_UNDO} };
16 struct sembuf decrease[] = { { 0, -1, SEM_UNDO}, {1, -1, SEM_UNDO} };
17
18 unsigned short seminitval[] = {1, 2};
19
```

- 13 int semid;
 - 세마포어 ID

- 15 ~ 16번 줄
 - 세마포어 조작

- 18 unsigned short seminitval[] = {1, 2};
 - 초기값 ok 1개, no 2개

```

20 union semun{
21     int val;
22     struct semid_ds *buf;
23     unsigned short int *array;
24     struct seminfo *__buf;
25 }semarg;
26
27 void errquit(char *message){
28     perror(message);
29     exit(0);
30 }
31
32 void do_work()
33 {
34     int count = 0;
35 #define Semop(val) if((semop val)==-1) errquit("semop")
36     while(count < 3)
37     {
38         Semop((semid, &decrease[ok],1));
39         printf("[pid : %5d] okWn", getpid());
40         Semop((semid, &decrease[no],1));
41         printf("[pid : %5d] noWn", getpid());
42         Semop((semid, &increase[ok],1));
43         printf("[pid : %5d] okWn", getpid());
44         Semop((semid, &increase[no],1));
45         printf("[pid : %5d] noWn", getpid());
46
47         sleep(1);
48         count++;
49     }
50 }
51

```

■ 32 ~ 50번 줄

- 각 프로세스가 수행할 작업

```

52 int main(int argc, char *argv[])
53 {
54     semid = semget(0x1234, 2, IPC_CREAT | 0600);
55     if(semid == -1)
56         semid = semget(0x1234, 0, 0);
57
58     semarg.array = seminitval;
59     if(semctl(semid, 0, SETALL, semarg) == -1)
60         errquit("semctl");
61
62     setvbuf(stdout, NULL, _IONBF, 0);
63
64     fork();
65     fork();
66     do_work();
67     semctl(semid, 0, IPC_RMID, 0);
68     return 0;
69 }

```

- 58 `semarg.array = seminitval;`
 - 세마포어 값 초기화

- 62 `setvbuf(stdout, NULL, _IONBF, 0);`
 - 표준 출력 non-buffering

- 64 ~ 65번 줄
 - 총 4개의 프로세스 생성

- 67 `semctl(semid, 0, IPC_RMID, 0);`
 - 세마포어의 삭제

실행 화면

```
$ ok_no
[pid : 11667] ok
[pid : 11667] no
[pid : 11667] ok
[pid : 11667] no
[pid : 11666] ok
[pid : 11666] no
[pid : 11666] ok
[pid : 11666] no
[pid : 11668] ok
[pid : 11668] no
[pid : 11668] ok
[pid : 11668] no
[pid : 11665] ok
[pid : 11665] no
[pid : 11665] ok
[pid : 11665] no
```

실행

- 파일명 : ok_no.c
- 컴파일 : gcc -o ok_no ok_no.c
- 사용법 : ok_no
- 동작설명
 - 이 프로그램은 세마포어의 이용 예를 보이는 최소한의 코드임
 - IPC_RMID에 의해 세마포어를 삭제하는 코드는 4개의 프로세스에 의해 4번 실행
→ 실제로 3번은 실패
 - do_work()실행 동안 다른 프로세스에 의해 세마포어가 삭제될 수 있음

8.5.4 공유메모리의 동기화문제 처리

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <sys/ipc.h>
7 #include <sys/shm.h>
8 #include <sys/sem.h>
9 #include <sys/types.h>
10
11 struct sembuf waitsem[] = { { 0, -1, 0 } };
12 struct sembuf notifysem[] = { { 0, +1, 0 } };
13
14 char *shm_data;
15 int shmid, semid;
16
17 #define Semop(val) W
18 { if(semop val == -1) W
19     errquit("semop fail");W
20 }
21
```

- 11 ~ 12번 줄
 - 세마포어 값을 조절하는 sembuf

- 14 char *shm_data;
 - 공유메모리에 대한 포인터

- 15 int shmid, semid;
 - 공유메모리와 세마포어 ID

- 17 ~ 20번 줄
 - semop함수 wrapper 매크로

```

22 union semun{
23     int val;
24     struct semid_ds *buf;
25     unsigned short int *array;
26     struct semifo *__buf;
27 }semarg;
28
29 void errquit(char *message)
30 {
31     perror(message);
32     exit(1);
33 }
34
35 void fork_and_run()
36 {
37     pid_t pid = fork();
38     if(pid<0)
39         errquit("fork() 실패");
40     else if(pid == 0)
41     {
42         busy();
43         exit(0);
44     }
45     return;
46 }
47

```

■ 35 ~ 46번 줄

- 자식 프로세스를 생성하여 busy() 함수 호출

■ 42 busy();

- 자식 프로세스가 실행

```

48 int busy()
49 {
50     int i = 0;
51     for(i=0 ; i<100 ; i++)
52     {
53         Semop((semid, &waitsem[0], 1));
54         access_shm();
55         Semop((semid, &notifysem[0], 1));
56     }
57     shmdt(shm_data);
58
59     return 0;
60 }
61
62 int access_shm()
63 {
64     int i;
65     pid_t pid;
66     struct timespec ts;
67     ts.tv_sec = 0;
68     ts.tv_nsec = 100000000;
69     sprintf(shm_data, "%d", getpid());
70
71     for(i=0 ; i<1000 ; i++)
72
73     pid=atoi(shm_data);
74     if(pid != getpid())
75         puts("Error : 다른 프로세스가 공유메모리에 접근함\n");
76     else
77     {
78         printf("ok\n");
79     }

```

- 48 ~ 60번 줄
 - 각 프로세스가 공유메모리에 접근하는 함수
- 51 ~ 56번 줄
 - 총 100번 접근
- 57 shmdt(shm_data);
 - 공유메모리 분리
- 62 ~ 85번 줄
 - 공유메모리에 접근
- 68 ts.tv_nsec = 100000000;
 - 0.1초
- 69 sprintf(shm_data, "%d", getpid());
 - 공유메모리에 자신의 pid를 기록
- 74 ~ 79번 줄
 - 공유메모리에 기록한 pid가 자신의 pid가 아니면 Error출력, 맞으면 ok 출력

```

81     fflush(stdout);
82     nanosleep(&ts, NULL);
83
84     return 0;
85 }
86
87 int main(int argc, char *argv[])
88 {
89     key_t shmkey, semkey;
90     unsigned short initsemval[1];
91     if(argc < 2)
92     {
93         printf("Usage : %s shmkey semkeyWn", argv[0]);
94         exit(1);
95     }
96
97     shmkey = atoi(argv[1]);
98     semkey = atoi(argv[2]);
99
100    shmkey = shmget(shmkey, 128, IPC_CREAT | 0600);
101    if(shmkey < 0)
102        errquit("shmget 실패");
103
104    shm_data = (char *)shmat(shmkey, (void*)0, 0);
105
106    if(shm_data == (char*)-1)
107        errquit("shmat 실패");
108

```

- 82 nanosleep(&ts, NULL);
 - sleep 설정

- 89 key_t shmkey, semkey;
 - 공유메모리, 세마포어 키

- 100 shmkey = shmget(shmkey, 128, IPC_CREAT | 0600);
 - 공유메모리 생성

- 104 shm_data = (char *)shmat(shmkey, (void*)0, 0);
 - 공유메모리 첨부

```

109     semid = semget(semkey, 1, IPC_CREAT | 0600);
110
111     if(semid == -1)
112         errquit("semget 실패");
113
114     initsemval[0] = 1;
115     semarg.array = initsemval;
116
117     if(semctl(semid,0,SETALL, semarg)==-1)
118         errquit("semctl");
119
120     fork_and_run();
121     fork_and_run();
122
123     busy();
124
125     wait(NULL);
126     wait(NULL);
127
128     shmctl(shmid, IPC_RMID, 0);
129     semctl(semid, 0, IPC_RMID, 0);
130     return 0;
131 }

```

- 109 `semid = semget(semkey, 1, IPC_CREAT | 0600);`
 - 세마포어 생성

- 114 `initsemval[0] = 1;`
 - 세마포어의 초기 값을 1로 설정
 - 공유메모리에 한 프로세스만 접근 가능

- 120 ~ 121번 줄
 - 자식 프로세스 생성

- 123 `busy();`
 - 부모프로세스의 공유메모리 접근

- 125 ~ 126번 줄
 - 자식 프로세스가 끝나기를 기다림

- 128 `shmctl(shmid, IPC_RMID, 0);`
 - 공유메모리 삭제

- 129 `semctl(semid, 0, IPC_RMID, 0);`
 - 세마포어 삭제

실행 화면

```
$ shm_control 1234 5555
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
```

실행

- 파일명 : shm_control.c
- 컴파일 : gcc -o shm_control shm_control.c
- 사용법 : shm_control [공유메모리 key] [세마포어 key]
- 동작설명
 - 공유메모리의 동기화문제를 세마포어를 사용해 해결

9장

스레드 프로그래밍

- 9.1 스레드의 생성과 종료
- 9.2 스레드 동기화
- 9.3 스레드간 통신
- 9.4 멀티스레드 에코 서버 프로그램

Overview

- Posix 표준 스레드 프로그래밍 기준의 스레드 처리방법 습득
- 스레드 생성, 동기화 처리, 뮤텍스, 조건변수, 세마포어 사용, 스레드 시그널 소개

9.1 스레드의 생성과 종료

스레드의 정의

프로세스처럼 독립적으로 수행되는 프로그램 코드

스레드의 정의

- 경량 프로세스
 - 프로세스 내에서 독립적으로 수행되는 제어의 흐름
- 한 프로세스 내에서 생성된 스레드들은 서로 전역 변수를 공유
 - 스레드간에 데이터를 편리하게 공유
 - 멀티프로세스 프로그램과 달리 IPC 기능을 사용하지 않아도 스레드간 데이터 공유

9.1.1 스레드 생성과 종료

pthread_create()

스레드 생성 함수

사용 문법

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
```

pthread_create()

- pthread_t *thread
 - 생성된 스레드 ID 리턴하며 이 ID를 통해 스레드에 접근
- pthread_attr_t *attr
 - 스레드 속성을 지정하는 인자
 - 디폴트로 사용하려면 NULL 지정
- void *(*start_routine)(void *)
 - 스레드 시작 함수(start routine)
- void *arg
 - 스레드 start routine 함수의 인자
- pthread_create() 호출시
 - start_routine 인자로 지정한 함수 실행
 - 스레드 시작 함수는 void* 타입의 인자를 받거나 임의의 타입을 가리키는 포인터를 리턴
 - 성공시 0, 실패시 에러코드를 정수 값으로 리턴
 - 시스템 자원부족, 최대 생성 스레드 수(PTHREAD_THREADS_MAX)를 넘을 때 EAGAIN 리턴

pthread_self()

스레드 자신의 스레드 ID 얻음

사용 문법

```
pthread_t pthread_self(void);
```

pthread_exit()

스레드가 스스로 종료

사용 문법

```
void pthread_exit(void *retval);
```

pthread_self()

- 스레드 자신의 ID를 얻음

pthread_exit()

- 스레드 종료 방법
 - 스레드 시작 함수내에서 return을 만남
 - 임의의 위치에서 pthread_exit()호출
- main()함수
 - 일종의 스레드로 main 스레드 또는 initial 스레드라 함
 - main()에서의 return은 다른 스레드에서의 return과 달리 프로세스를 종료시킴
 - main()에서 여러 스레드를 생성한 후 return하면 모두 종료됨
 - main()에서 exit()를 호출하거나 return하는 대신 pthread_exit()를 호출하는 경우
 - main 스레드만 종료하며 생성된 스레드들은 종료되지 않음
 - 생성된 모든 스레드가 종료될 때까지 프로세스는 종료되지 않고 기다림

pthread_join()

자신이 생성한 자식 스레드가 종료할 때까지 기다려야 하는 경우에 사용

사용 문법

```
int pthread_join(pthread_t thrd, void **thread_return);
```

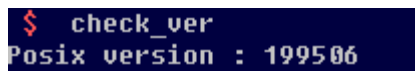
pthread_join()

- thrd 인자
 - 종료를 기다리는 스레드의 ID
 - 지정한 스레드가 종료할 때까지 또는 취소될 때까지 대기
- thread_return 인자
 - 자식 스레드의 종료 상태가 저장
 - NULL로 지정할 경우 스레드 종료 상태 값을 받지 않음
 - 자식 스레드의 종료 값은 자식 스레드에서 스레드 종료 시에 pthread_exit()의 인자 또는 return 값으로 남길수 있음
 - 자식 스레드가 다른 스레드에 의해 취소된 경우
 - 스레드 종료 값은 PTHREAD_CANCELED
- 자식 스레드가 종료되기 전에 부모 스레드가 먼저 종료 되지 않도록 할 경우에 사용
 - 자식 스레드의 종료 시점을 정확히 파악하여 다른 작업을 할 경우에 사용
 - 자식 스레드의 종료 상태 값을 얻기 위해 사용
- main 스레드에서 주로 사용하게 됨
 - main에서 생성된 스레드들이 종료하기 전에 main이 종료되면 생성된 스레드가 모두 종료됨

Posix 스레드 라이브러리

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     long version = sysconf(_SC_VERSION);
8     printf("Posix version : %ld\n", version);
9     return 0;
10 }
```

실행 화면



```
$ check_ver
Posix version : 199506
```

Posix 스레드 라이브러리

- 스레드 프로그램을 작성하기 위한 스레드 라이브러리

실행

- 파일명 : check_ver.c
- 컴파일 : gcc -o check_ver check_ver.c
- 사용법 : check_ver
- 동작설명
 - version이 199506 이상인 시스템에서는 Posix 라이브러리 사용이 가능
 - version이 199506 미만인 시스템에서는 Posix1003.1c 스레드를 지원하지 않음

스레드의 생성과 종료

```
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 char name[10];
8
9 void *thrfunc(void *arg)
10 {
11     printf("(%s' thread routine) Process ID = %d\n", name, getpid());
12     printf("(%s' thread routine) Thread ID = %d\n", name, pthread_self());
13 }
14
15 int main(int argc, char **argv)
16 {
```

- 9 ~ 13번 줄
 - 스레드 시작 함수

```

17  int status;
18  pthread_t tid;
19  pid_t pid;
20
21  pid = fork();
22
23  if(pid == 0)
24      sprintf(name, "child");
25  else
26      sprintf(name, "parent");
27
28  printf("(%s's main) Process ID = %d\n", name, getpid());
29  printf("(%s's main) Init thread ID = %d\n", name, pthread_self());
30
31  if((status = pthread_create(&tid, NULL, &thrfunc, NULL)) != 0)
32  {
33      printf("스레드 생성 실패 : %s\n", strerror(status));
34      exit(0);
35  }
36
37  pthread_join(tid, NULL);
38  printf("\n%s [%d] 스레드 종료\n", name, tid);
39  return 0;
40 }

```

- 21 pid = fork();
 - 자식 프로세스 생성

- 28 ~ 29번 줄
 - 프로세스 ID와 초기 스레드의 ID 확인

- 31 ~ 35번 줄
 - 에러 발생 시 에러코드를 리턴

- 37 pthread_join(tid, NULL);
 - 인자로 지정한 스레드 ID가 종료하기를 기다림

실행 화면

```
$ thread_start
(child's main) Process ID = 12763
(child's main) Init thread ID = 1073971840
(parent's main) Process ID = 12762
(parent's main) Init thread ID = 1073971840
(child' thread routine) Process ID = 12763
(child' thread routine) Thread ID = 1082363072

child [1082363072] 스레드 종료
(parent' thread routine) Process ID = 12762
(parent' thread routine) Thread ID = 1082363072

parent [1082363072] 스레드 종료
```

실행

- 파일명 : thread_start.c
- 컴파일 : gcc -o thread_start thread_start.c -lpthread
- 사용법 : thread_start
- 동작설명
 - pthread_create()와 pthread_self()함수를 이용해 스레드를 생성하고 스레드 ID를 얻음
 - 컴파일시 “-lpthread” 로 pthread 라이브러리를 링크

REENTRANT

일반 함수들이 멀티스레드 환경에 적합하게 동작하게 해줌

REENTRANT

- 스레드 프로그램 컴파일시 -D_REENTRANT 옵션
 - 프로그램에서 #define _REENTRANT 를 선언한 효과
- 일반 함수에서 errno 값을 이용할 경우
 - 스레드 함수에서는 errno 값을 사용하지 않아야 함
 - 이 문제를 피하기 위해 errno를 전역 변수가 아닌 각 스레드별 고유의 변수로 사용
 - 컴파일 시 -D_REENTRANT 옵션으로 _REENTRANT가 설정되어 스레드들은 고유한 errno를 가짐
 - 일반 함수 호출 시 errno 변수나 perror() 함수를 사용 가능하게 함

9.1.2 스레드의 상태

스레드의 상태

준비, 실행, 블록, 종료 중 하나를 가지게 됨

스레드의 상태

- 준비
 - 처음 생성 시 가지게 되는 상태
 - 스레드가 실행될 수 있는 상태
- 실행
 - 운영체제의 스케줄링에 의해 이동
 - CPU의 서비스를 받고 있는 상태
- 블록
 - 즉시 처리할 수 없는 작업을 만날 경우
 - `sleep()`, `read()`, 세마포어 연산 등으로 기다리는 상태
- 종료
 - 스레드 시작 함수에서 `return`하거나 `pthread_exit()` 호출, 다른 스레드에 의해 취소될 경우
 - 스레드가 종료 또는 취소된 상태

스레드의 분류

joinable 스레드와 분리된 스레드로 나뉨

스레드의 분류

- 스레드에서 pthread_detach()를 호출 시
 - 데몬 프로세스처럼 부모 스레드와 분리된 스레드로서 실행됨
 - 분리된 스레드는 pthread_join()을 호출할 수 없고 종료시 종료 상태에 남아 있지 않음
→ 메모리가 모두 반환됨

- 모든 스레드는 디폴트로 joinable 스레드
 - 부모 스레드는 pthread_join()을 호출하여 이 스레드의 종료를 기다림

9.2 스레드 동기화

9.2.1 동기화 문제

동기화 문제

동기화 문제 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 #define MAX_THR 2
8
9 int run = -1;
10
11 void *thrfunc(void *arg)
12 {
13     while(1)
14     {
15         prn_data(pthread_self());
16     }
17     return NULL;
18 }
19
```

동기화 문제

- 한 스레드가 공유 데이터를 액세스하는 도중에 다른 스레드가 이 공유 데이터 액세스 시
 - 데이터 값을 두 스레드가 정확히 예측할 수 없음
- 해결 방법
 - 스레드들이 공유데이터에 접근할 때 서로 배타적으로 접근
 - 한 번에 한 스레드만 공유데이터에 접근
 - 뮤텝스(mutex = mutual + exclusion) 사용

동기화 문제 예

- 9 int run = -1;
 - prn_data() 수행중인 스레드 ID
 - 초기 값 -1
- 11 ~ 18번 줄
 - 스레드 시작 함수

```

20 void prn_data(long run_thread)
21 {
22     run = run_thread;
23     if(run != pthread_self())
24     {
25         printf("Error : %d 스레드 실행중 run = %d\n", run_thread, run);
26     }
27     run = -1;
28 }
29
30 int main(int argc, char **argv)
31 {
32     pthread_t tid[MAX_THR];
33     int i, status;
34     for(i=0 ; i<MAX_THR ; i++)
35     {
36         if((status = pthread_create(&tid[i], NULL, &thrfunc, NULL)) != 0)
37         {
38             printf("스레드 생성 실패 : %s\n", strerror(status));
39             exit(0);
40         }
41     }
42     pthread_join(tid[0], NULL);
43
44     return 0;
45 }

```

■ 20 ~ 28번 줄

- 스레드 ID 출력

■ 27 run = -1;

- 초기 값으로 환원

실행 화면

```
$ thread_syn
Error : 1082363072 스레드 실행 중 run = 1090751552
Error : 1082363072 스레드 실행 중 run = -1
Error : 1082363072 스레드 실행 중 run = -1
Error : 1090751552 스레드 실행 중 run = 1082363072
Error : 1082363072 스레드 실행 중 run = -1
Error : 1082363072 스레드 실행 중 run = -1
```

실행

- 파일명 : thread_syn.c
- 컴파일 : gcc -o thread_syn thread_syn.c -lpthread
- 사용법 : thread_syn
- 동작설명
 - 스레드들이 스레드 ID를 저장하는 run 변수를 공유
 - 한 스레드가 prn_data()를 호출하는 동안 다른 스레드가 prn_data() 호출시 run 값 변경
 - 중간에 run값이 변경될 경우 에러 메시지를 출력함

플래그 사용 예

thrfunc()

```
1 void *thrfunc(void *arg)
2 {
3     while(1)
4     {
5         if(run == -1)
6         {
7             prn_data(pthread_self());
8         }
9     }
10    return NULL;
11 }
```

플래그 사용 예

- prn_data() 함수에서 동기화 문제 발생
 - 위의 thread_syn.c 프로그램에서 prn_data() 함수가 동시에 호출되므로 동기화 문제 발생
- 스레드 시작 함수 thrfunc()를 수정
 - 플래그를 하나 정의하여 사용하므로써 한 스레드만 prn_data()를 실행하도록 수정
 - run == -1 인 경우, 즉 아무 스레드도 prn_data()를 호출하지 않을 때만 prn_data() 호출
 - 플래그를 조사한 직후에 스레드 스케줄링이 일어날 경우
 - 동기화 문제 발생

9.2.2 뮷텍스

뮷텍스

스레드의 동기화 문제를 해결하는 커널이 제공하는 일종의 플래그

뮷텍스 사용 방법

뮷텍스 잠금과 해제

```
1 pthread_mutex_t mutex;  
2  
3 pthread_mutex_lock(mutex);  
4  
5 pthread_mutex_unlock(mutex);
```

뮷텍스 사용 방법

- 뮷텍스 잠금
 - 스레드가 공유데이터를 사용하기 전, 즉 크리티컬 영역에 들어가기 전에 잠금
 - 뮷텍스 잠금을 한 스레드만 공유데이터에 접근
- 뮷텍스 해제
 - 스레드가 크리티컬 영역을 나올 때 뮷텍스 잠금을 해제
 - 다른 스레드는 뮷텍스 잠금이 해제될 때까지 대기
- 1 pthread_mutex_t mutex;
 - 뮷텍스 선언
- 3 pthread_mutex_lock(mutex);
 - 뮷텍스 잠금
- 5 pthread_mutex_unlock(mutex);
 - 뮷텍스 해제

무텍스 사용

thread_syn.c 프로그램의 thrfunc()를 수정

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3 void *thrfunc(void *arg)
4 {
5     while(1)
6     {
7         pthread_mutex_lock(&lock);
8         prn_data(pthread_self());
9         pthread_mutex_unlock(&lock);
10    }
11    return NULL;
12 }
```

- 1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
 - 무텍스 선언 및 초기화

- 7 pthread_mutex_lock(&lock);
 - 무텍스 잠금

- 8 prn_data(pthread_self());
 - 공유데이터 액세스

- 9 pthread_mutex_unlock(&lock);
 - 무텍스 해제

뮤텍스 선언 및 초기화

뮤텍스 초기화

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_init(&mutex, &attr);
```

뮤텍스 선언 및 초기화

■ 뮤텍스 선언

- 뮤텍스의 변수 타입 이름은 pthread_mutex_t
- 여러 스레드들이 사용하므로 전역 변수로 선언

■ 뮤텍스 초기화

- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
→ 매크로를 이용한 초기화 방법으로 기본 속성만 가짐
- pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
→ 함수를 이용한 초기화 방법으로 attr 인자를 이용해 속성을 지정함
→ attr 인자가 NULL 일 경우 기본 속성이 적용됨

■ pthread_mutexattr_settype()

- attr 인자의 속성을 지정
- 인자로 &attr 과 type가 있음
- type은 뮤텍스의 타입을 지정하기 위한 상수
 - PTHREAD_MUTEX_TIMED_NP : timed 타입
 - PTHREAD_MUTEX_RECURSIVE_NP : recursive 타입
 - PTHREAD_MUTEX_ERRORCHECK_NP : error checking 타입

기본 뮤텍스

특별한 타입을 지정하지 않은 뮤텍스

Timed 타입 뮤텍스

지정 시간 동안만 블록

기본 뮤텍스

- 단 한 번의 잠금만 허용
 - 1번 스레드가 뮤텍스 잠금을 한 상황에서 2번 스레드가 잠금 시도시
→ 2번 스레드는 1번 스레드의 잠금이 해제될 때까지 대기
 - 1번 스레드에 의해 잠긴 뮤텍스에 대해 1번 스레드가 또 잠금 시도시
→ 데드락이 발생하여 1번 스레드는 영원히 블록 상태가 됨
→ pthread_mutex_lock() 대신 pthread_mutex_trylock()를 사용하여 블록 현상 피함
→ 뮤텍스를 얻지 못하는 경우 스레드는 블록되지 않고 리턴되며 EBUSY 에러 발생
- 다른 스레드에서 해제 가능
 - 1번 스레드에 의해 잠금된 뮤텍스를 2번 스레드에서 해제 가능

Timed 타입 뮤텍스

- pthread_mutex_timedlock()
 - pthread_mutex_lock()는 뮤텍스를 얻을 때까지 무한 대기
 - timed 타입 뮤텍스로 지정된 시간 동안만 블록
 - 리눅스에서 기본 뮤텍스가 timed 타입 뮤텍스로 동작함

Recursive 타입 뮷텍스

한 스레드가 한 번 이상 잠금

Error Check 타입 뮷텍스

두 번 잠금 시도 시 에러 발생 후 프로세스 종료

Recursive 타입 뮷텍스

- pthread_mutex_lock()을 호출한 횟수만큼 pthread_mutex_unlock()를 호출
 - 한 스레드가 한 번 이상의 잠금을 하므로 호출 횟수만큼 해제해야 잠금이 해제됨
 - 한 스레드가 잠금을 한 상태에서 다른 스레드가 잠금 해제를 시도 시 에러 발생

Error Check 타입 뮷텍스

- 기본 뮷텍스에서 한 스레드가 뮷텍스 잠금을 두 번 시도 시
 - 무한 블록 상태
- error check 타입 뮷텍스에서 두 번 잠금 시도 시
 - 에러 발생 후 프로세스 종료
- error check 타입 뮷텍스에서 잠겨 있지 않은 뮷텍스에 잠금 해제 시도 시
 - 에러 발생 후 프로세스 종료
 - 기본 뮷텍스에서는 잠겨있지 않은 뮷텍스에 잠금 해제 시도 시 에러 없음
- 자체적으로 에러 검사를 하므로 기본 뮷텍스보다 처리 속도가 느림

뮤텍스 삭제

뮤텍스를 더 이상 사용하지 않을 때 사용

사용 방법

```
pthread_mutex_unlock(&mutex);  
pthread_mutex_destroy(&mutex);
```

데드락

두 개의 스레드가 뮤텍스 해제를 서로 기다리는 현상

뮤텍스 삭제

- 뮤텍스가 어떤 스레드에 의해 잠금된 상태에서 제거 시도 시
 - EBUSY 에러 발생
 - 뮤텍스의 현재 상태를 알아보는 방법이 없음
- 안전한 뮤텍스 제거를 위해 무조건 뮤텍스 해제 후 제거

데드락

- 두 개 이상의 스레드가 두 개 이상의 뮤텍스를 사용하는 경우에 발생
 - 각 스레드가 뮤텍스를 하나씩 잠그고 있는 상태에서 상대방의 뮤텍스 해제를 서로 기다림
- 데드락 발생시 프로그램은 영원히 블록됨
- 서로 연관이 있는 작업에서는 다수의 뮤텍스 사용을 피함
 - 작업 처리의 순서를 정하여 해결
- 뮤텍스를 많이 사용하면 프로그램 성능이 저하됨
 - 뮤텍스를 얻은 상태에서 처리하는 작업량을 최소화하여 해결
 - 뮤텍스가 잠긴 동안 다른 스레드가 블록될 확률이 높아서 크리티컬 영역이 길수록 전체 성능이 저하됨

9.2.3 뮤텍스 사용 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6
7 int count = 0;
8 pthread_mutex_t count_lock;
9 pthread_mutexattr_t mutex_attr;
10
11 void *thrfunc(void *arg)
12 {
13     while(1)
14     {
15         pthread_mutex_lock(&count_lock);
16         printf("[%ld 스레드] 뮤텍스 잠금 Wn", pthread_self());
17         printf("[%ld 스레드] count = %dWn", pthread_self(),count);
18         count++;
19         sleep(1);
20         printf("[%ld 스레드] 뮤텍스 해제 WnWn",pthread_self());
21         pthread_mutex_unlock(&count_lock);
22     }
23     return NULL;
24 }
25
```

- 7 int count = 0;
 - 공유 데이터

- 8 pthread_mutex_t count_lock;
 - 뮤텍스 초기화

- 9 pthread_mutexattr_t mutex_attr;
 - 뮤텍스 속성 초기화

- 11 ~ 24번 줄
 - 스레드 시작 함수

```

26 int main(int argc, char **argv)
27 {
28     pthread_t tid[2];
29     int i, status;
30     pthread_mutexattr_init(&mutex_attr);
31     pthread_mutex_init(&count_lock, &mutex_attr);
32     for(i=0; i<2 ; i++)
33     {
34         if((status=pthread_create(&tid[i], NULL, &thrfunc, NULL))!= 0)
35         {
36             printf("스레드 생성 실패 : %s", strerror(status));
37             exit(0);
38         }
39     }
40
41     for(i=0 ; i<2 ; i++)
42         pthread_join(tid[i], NULL);
43
44     return 0;
45 }

```

- 30 pthread_mutexattr_init(&mutex_attr);
 - 뮤텝스

- 31 pthread_mutex_init(&count_lock, &mutex_attr);
 - 뮤텝스 속성 변수

실행 화면

```
$ mutex_example
[1082363072 스레드] 뮉텍스 잠금
[1082363072 스레드] count = 0
[1082363072 스레드] 뮉텍스 해제

[1082363072 스레드] 뮉텍스 잠금
[1082363072 스레드] count = 1
[1082363072 스레드] 뮉텍스 해제

[1082363072 스레드] 뮉텍스 잠금
[1082363072 스레드] count = 2
[1082363072 스레드] 뮉텍스 해제
```

실행

- 파일명 : mutex_example.c
- 컴파일 : gcc -o mutex_example mutex_example.c -lpthread
- 사용법 : mutex_example
- 동작설명
 - 한 스레드가 count 변수에 접근하는 동안 다른 스레드의 접근을 못하게 함
 - main()에서 두 개의 스레드를 만들고 스레드 시작 함수에서 뮉텍스를 이용해 1초 간격으로 count 값을 출력

9.3 스레드간 통신

9.3.1 조건변수 사용 방법

조건변수의 정의

스레드간에 공유데이터의 상태에 대한 정보를 주고받는 변수

조건변수의 정의

■ 조건변수의 사용 예

- 생산자 스레드는 큐에 메시지를 쓰고 소비자 스레드는 큐의 메시지를 꺼내서 출력
→ 큐는 스레드간 공유데이터
- 동기화 문제를 피하기 위해 뮤텍스를 사용
- 운영체제의 스케줄러가 생산자와 소비자의 두 스레드가 교대로 실행함을 보장하지 않음
→ 어느 한 스레드가 연속으로 두 번 실행될 수 있음
- 조건변수를 이용해 큐에 메시지가 쓰였음을 소비자 스레드에 알려줌

■ 조건알림

- 조건변수를 통해 어떤 조건이 만족됨을 다른 스레드에 알려주는 기능

조건변수의 동작

뮤텍스를 정의한 후 뮤텍스와 연계하여 조건변수를 정의

사용 문법

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

조건변수의 동작

- 반드시 뮤텍스와 함께 사용
- pthread_cond_wait()
 - 소비자 스레드에서 조건변수를 기다리는 함수
 - 조건변수 cond 인자
 - 조건알림이 발생할 때까지 대기
 - 뮤텍스 타입의 mutex 인자
 - 조건변수를 제어하는 인자
- pthread_cond_wait() 호출시 내부 동작
 - 뮤텍스해제 → 블록 상태 → 조건알림을 받음 → 깨어나면서 뮤텍스를 얻음
- pthread_cond_timedwait()
 - 지정한 시간동안만 블록 상태에서 조건알림을 기다림
 - 조건알림을 무한히 기다리는 것을 방지

pthread_cond_signal()

조건알림을 기다리는 스레드에게 조건알림을 보냄

사용 문법

```
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_broadcast()

여러 스레드를 동시에 깨움

pthread_cond_signal()

- 조건알림을 기다리는 스레드를 블록 상태에서 깨어나게 함

pthread_cond_broadcast()

- 생산자 스레드보다 소비자 스레드가 많은 경우
 - pthread_cond_signal()은 한 번에 하나의 스레드만 깨움
 - 조건알림을 기다리는 스레드가 둘 이상일 때 모든 wait 함수를 깨우기 위해 사용
- 여러 스레드를 동시에 깨우지만 공유데이터 접근을 위해서는 뮤텝스를 얻어야 함

조건변수의 생성과 삭제

pthread_cond_init()를 이용해 초기화

```
1 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
2  
3 pthread_cond_t cond;  
4 pthread_cond_init(&cond, NULL);
```

pthread_cond_destroy()를 이용해 삭제

```
pthread_cond_destroy(&cond);
```

조건변수의 생성과 삭제

■ 조건변수 초기화

- pthread_cond_init()를 이용해 사용전 초기화
- 1 pthread_cond_t cond = PTHREAD_COND_INITIALIZER
→ 매크로를 이용한 초기화 방법
- 3 ~ 4번 줄
→ 함수를 이용한 초기화 방법으로 3번줄에서 조건변수 선언
→ &cond는 초기화 시킬 조건변수
→ 두 번째 인자는 조건변수의 속성으로 NULL 일 경우 기본 속성으로 초기화

■ 조건변수 삭제

- pthread_cond_destroy()를 이용해 삭제

9.3.2 조건변수 사용 예

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <pthread.h>
7
8 int sleep_time_time;
9
10 typedef struct _complex{
11     pthread_mutex_t mutex;
12     pthread_cond_t cond;
13     int value;
14 }thread_control_t;
15
16 thread_control_t data={PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
17
18 void thr_errquit(char *message, int errcode)
19 {
20     printf("%s: %s\n", message, strerror(errcode));
21     pthread_exit(NULL);
22 }
23
```

- 8 int sleep_time;
 - 자식 스레드가 처음 잠자는 시간

- 10 ~ 14번 줄
 - 뮤텍스, 조건변수, 공유데이터

- 16 thread_control_t data={PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
 - 뮤텍스, 조건변수, 공유데이터의 초기화

- 18 ~ 22번 줄
 - 에러 출력 및 스레드 종료

```

24 void *wait_thread(void *arg)
25 {
26     int status;
27     sleep(sleep_time);
28     if((status=pthread_mutex_lock(&data.mutex)) != 0)
29         thr_errquit("pthread_mutex_lock failure",status);
30
31     data.value = 1;
32     if((status=pthread_cond_signal(&data.cond)) != 0)
33         thr_errquit("pthread_cond_signalure",status);
34
35     if((status=pthread_mutex_unlock(&data.mutex)) != 0)
36         thr_errquit("pthread_mutex_unlock failure",status);
37
38     return NULL;
39 }
40
41 int main(int argc,char **argv){
42     int status;
43     pthread_t wait_thr;
44     struct timespec timeout;
45     if(argc != 2)
46     {
47         printf("사용법 : condition_test [자식 스레드가 sleep할 시간]\n");
48         exit(0);
49     }
50     sleep_time = atoi(argv[1]);
51     if((status = pthread_create(&wait_thr, NULL, wait_thread, NULL)) != 0)
52     {
53         printf("스레드 생성 실패 : %s\n", strerror(status));
54         exit(1);
55     }

```

■ 24 ~ 39번 줄

- 스레드 시작 함수

```

57     timeout.tv_sec = time(NULL) + 5;
58     timeout.tv_nsec = 0;
59
60     if((status = pthread_mutex_lock(&data.mutex))!=0)
61     {
62         printf("뮤텍스 잠금 실패 : %s\n", strerror(status));
63         exit(1);
64     }
65
66     if(data.value == 0)
67     {
68         status = pthread_cond_timedwait(&data.cond, &data.mutex, &timeout);
69         if(status == ETIMEDOUT)
70         {
71             printf("조건변수 대기 시간 초과\n");
72         }
73         else{
74             printf("조건변수 대기중\n");
75         }
76     }
77

```

- 57 `timeout.tv_sec = time(NULL) + 5;`
 - 현재의 시간 + 5초

- 60 `if((status = pthread_mutex_lock(&data.mutex))!=0)`
 - 뮤텍스 잠금

```

78     if(data.value == 1)
79         printf("조건변수 깨어남.Wn");
80     if((status = pthread_mutex_unlock(&data.mutex)) != 0)
81     {
82         printf("뮤텍스 해제 실패 : %sWn", strerror(status));
83         exit(0);
84     }
85
86     if((status=pthread_join(wait_thr, NULL)) != 0)
87     {
88         printf("pthread_join : %sWn", status);
89         exit(0);
90     }
91
92     return 0;
93 }

```

■ 86 ~ 90번 줄

- 자식 스레드가 종료하기를 기다림

실행 화면

```
$ condition_test 1
조건변수 대기중
조건변수 깨어남.
```

```
$ condition_test 5
조건변수 대기 시간 초과
```

실행

- 파일명 : condition_test.c
- 컴파일 : gcc -o condition_test condition_test.c -lpthread
- 사용법 : condition_test [자식 스레드가 sleep 할 시간]
- 동작설명
 - 공유데이터인 data.value의 초기 값은 0이나 부모 스레드는 지정도니 시간동안 이 값을 자식 스레드가 1로 설정해주기를 기다림
 - 자식 스레드는 시작 직후 인자로 넣어준 시간동안 sleep함
 - 부모 스레드는 pthread_cond_timedwait()를 호출하여 5초동안만 조건 알림을 기다림
 - 자식 스레드는 sleep()에서 깨어난 후 공유데이터 값을 1로 설정
 - 조건변수 data.cond를 통해 대기 상태인 부모 스레드에게 조건알림을 함
 - 자식 스레드가 5초이상 sleep하면 부모 스레드는 ETIMEDOUT 에러를 발생시킴

9.4 멀티스레드 에코 서버 프로그램

```
1 #include<stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <errno.h>
6 #include <pthread.h>
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <fcntl.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <sys/ipc.h>
13 #include <sys/msg.h>
14
15 #define MAX_BUFFER 1024
16
17 typedef struct _message{
18     long message_type;
19     struct sockaddr_in addr;
20     char message_text[MAX_BUFFER];
21 }message_t;
22
23 struct sockaddr_in servaddr;
24 int sock;
25 int msqid;
26
```

- 24 int sock;
 - 서버 소켓

- 25 int msqid;
 - 메시지큐 ID

```

27 void *echo_recv(void *arg)
28 {
29     int nbytes, status, len = sizeof(struct sockaddr);
30     message_t pmessage;
31     int size;
32     size = sizeof(pmessage) - sizeof(long);
33     pmessage.message_type = pthread_self();
34     while(1) {
35         nbytes = recvfrom(sock, pmessage.message_text, MAX_BUFFER, 0,
                           (struct sockaddr *)&pmessage.addr, &len);
36         if(nbytes < 0)
37             thr_errquit("recvfrom fail", errno);
38         pmessage.message_text[nbytes] = 0;
39         printf("recv thread = %ldWn", pthread_self());
40         if(msgsnd(msqid, &pmessage, size, 0) == -1)
41             thr_errquit("메시지보냄 실패", errno);
42     }
43 }
44
45 void *echo_resp(void *arg)
46 {
47     message_t pmessage;
48     int nbytes, len = sizeof(struct sockaddr);
49     int size;
50     size = sizeof(pmessage) - sizeof(long);
51     pmessage.message_type = 0;
52     while(1)
53     {
54         if(msggrcv( msqid,(void *)&pmessage, size, 0, 0) < 0)
55             {
56                 perror("msgrcv fail"); exit(0);
57             }

```

- 27 ~ 43번 줄
 - 에코 수신 스레드
- 45 ~ 72번 줄
 - 에코 송신 스레드
- 35 nbytes = recvfrom(sock, pmessage.message_text, MAX_BUFFER, 0,
 (struct sockaddr *)&pmessage.addr, &len);
 - 에코 메시지 수신
- 40 if(msgsnd(msqid, &pmessage, size, 0) == -1)
 - 메시지큐로 전송
- 54 if(msgrcv(msqid,(void *)&pmessage, size, 0, 0) < 0)
 - 메시지큐에서 읽음

```

58     nbytes = sendto(sock, pmessage.message_text, strlen(pmessage.message_text), 0,
59                     (struct sockaddr*)&pmessage.addr, len);
60     if(nbytes < 0)
61         thr_errquit("전송 실패",errno);
62
63     printf("응답 스레드 = %dWnWn", pthread_self());
64     pmessage.message_text[0] = 0;
65 }
66 }
67
68 void errquit(char *message)
69 {
70     perror(message);
71     exit(-1);
72 }
73
74 void thr_errquit(char *message, int errcode)
75 {
76     printf("%s : %sWn",message, strerror(errcode));
77     pthread_exit(NULL);
78 }
79
80 int main(int argc, char *argv[])
81 {
82     pthread_t tid[6];
83     struct sockaddr_in cliaddr;
84     int port, status, i, len = sizeof(struct sockaddr);
85     key_t msqkey;

```

- 58 ~ 59번 줄
- 에코 응답

```

86     if(argc!=3)
87     {
88         printf("Usage: %s msgkey portWn", argv[0]);
89         exit(1);
90     }
91     msqkey = atoi(argv[1]);
92     port = atoi(argv[2]);
93
94     if( (msqid = msgget(msqkey, IPC_CREAT | 0660)) < 0)
95         errquit("메시지큐 실패");
96
97     if( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
98         errquit("소켓 실패");
99
100    bzero(&servaddr, len);
101    servaddr.sin_port = htons(port);
102    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
103    servaddr.sin_family=AF_INET;
104    bind(sock, (struct sockaddr*)&servaddr, len);
105
106    for(i=0; i<5; i++)
107        if((status=pthread_create(&tid[i], NULL, echo_rcv, NULL))!=0)
108            thr_errquit("pthread_create", status);
109
110    if((status=pthread_create(&tid[5], NULL, echo_resp, NULL))!=0)
111        thr_errquit("pthread_create", status);
112    for(i=0; i<6; i++)
113        pthread_join(tid[i], NULL);
114
115    msgctl(msqid, IPC_RMID, 0);
116    return 0;
117 }

```

- 94 if((msqid = msgget(msqkey, IPC_CREAT | 0660)) < 0)
 - 메시지큐 생성

- 97 if((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 - 소켓 생성

- 115 msgctl(msqid, IPC_RMID, 0);
 - 메시지큐 삭제

실행 화면

서버 실행 화면

```
$ multithread_echoserv 1234 5555
recv thread = 1082363072
응답 스레드 = 1133718080

recv thread = 1099140032
응답 스레드 = 1133718080

recv thread = 1125329600
응답 스레드 = 1133718080
```

클라이언트 실행 화면

```
$ udp_echocli 210.115.49.220 test3 5555
Received: test3
```

실행

- 파일명 : multithread_echoserv.c
- 컴파일 : gcc -o multithread_echoserv multithread_echoserv.c -lpthread
- 사용법 : multithread_echoserv [메시지큐 키] [포트]
- 동작설명
 - 클라이언트 프로그램으로는 3장의 udp_echocli.c 프로그램을 사용
 - 수신 스레드는 클라이언트로부터 메시지를 수신하여 메시지큐에 넣음
 - 송신 스레드는 메시지큐에서 메시지를 꺼내 클라이언트로 응답