Network Programming
(Day 2)

Yunmin Go

School of CSEE

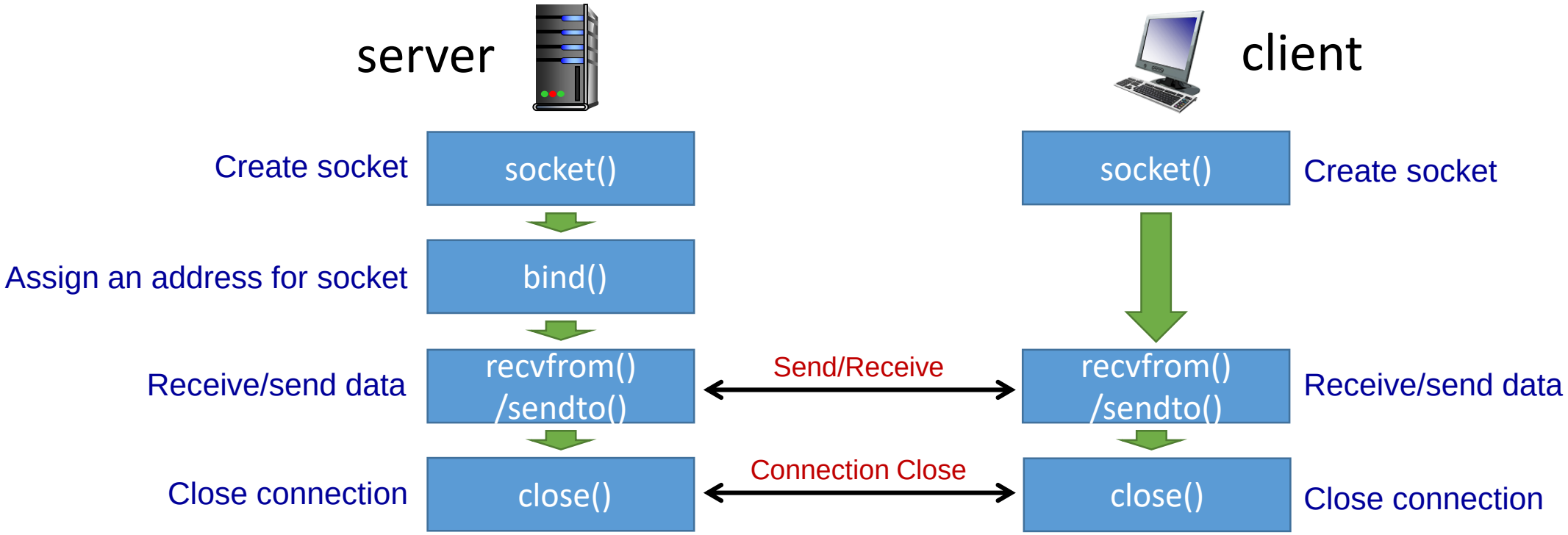
Handong Global University

HANDONG GLOBAL UNIVERSITY

Network Programming (Day 2): roadmap

- UDP Socket Programming
- Socket Options
- Multicast and Broadcast
- Multi-Process Programming #1
- Multi-Process Programming #2

UDP Server/Client Function Call



recvfrom()

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Receive a message from a socket

- Block until data received
- *sockfd*: socket file descriptor
- *buffer*: points to a buffer where the message should be stored
- *length*: receives the data up to *length* bytes into *buffer*
- *flags*: type of message reception
  - 0 for regular
- *src_addr*: pointer to a sockaddr structure to be filled in with the address of the peer socket
- *addrlen*: it will contain the actual size of the peer address
- Return value
  - Success: the number of bytes received
    - Zero for EOF
  - Error: -1
- Example

```
str_len = recvfrom(serv_sock, message, BUF_SIZE, 0,
                   (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
```

sendto()

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
                        const struct sockaddr *dest_addr, socklen_t *addrlen);
```

Send a message on a socket

- *sockfd*: socket file descriptor
- *buffer*: buffer containing the message to send
- *length*: transmits the data in *buffer* upto *length* bytes
- *flags*: type of message transmission
  - 0 for regular
- *dest_addr*: address to transmit data
- *addrlen*: the size (bytes) of the address structure pointed to by *addr*

- Return value
  - Success: the number of bytes sent
  - Error: -1
- Example

```
sendto(serv_sock, message, str_len, 0,
        (struct sockaddr*)&clnt_adr, clnt_adr_sz);
```

Example: uecho_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int serv_sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t clnt_adr_sz;

   struct sockaddr_in serv_adr, clnt_adr;
   if (argc != 2) {
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }
```

```
   serv_sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (serv_sock == -1)
      error_handling("UDP socket creation error");

   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));

   if (bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr)) == -1)
      error_handling("bind() error");

   while(1)
   {
      clnt_adr_sz = sizeof(clnt_adr);
      str_len = recvfrom(serv_sock, message, BUF_SIZE, 0,
                         (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
      sendto(serv_sock, message, str_len, 0,
                         (struct sockaddr*)&clnt_adr, clnt_adr_sz);
   }

   close(serv_sock);
   return 0;
}
```

socket()

bind()

recvfrom()/
sendto()

close()

Example: uecho_client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t adr_sz;

   struct sockaddr_in serv_adr, from_adr;
   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (sock == -1)
      error_handling("socket() error");

   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family=AF_INET;
   serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
   serv_adr.sin_port=htons(atoi(argv[2]));

   while(1)
   {
      fputs("Insert message(q to quit): ", stdout);
      fgets(message, sizeof(message), stdin);
      if (!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
         break;

      sendto(sock, message, strlen(message), 0,
            (struct sockaddr*)&serv_adr, sizeof(serv_adr));
      adr_sz = sizeof(from_adr);
      str_len = recvfrom(sock, message, BUF_SIZE, 0,
               (struct sockaddr*)&from_adr, &adr_sz);

      message[str_len]=0;
      printf("Message from server: %s", message);
   }

   close(sock);
   return 0;
}
```

socket()

sendto()/
recvfrom()

close()

Connected UDP

- Internal procedures in sendto()
  - 1) Allocate a destination IP and port for UDP socket
  - 2) Send data to allocated destination IP and port
  - 3) Deallocate destination IP and port from UDP socket
- Connected UDP
  - We can use the socket that has destination address already assigned
    - Skip address allocate and deallocate procedure
  - We can use write() and receive() instead of sendto() and recvfrom()
  - It does not mean connection-oriented socket like TCP

How to create connected UDP

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == -1)
   error_handling("socket() error");

memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family = AF_INET;
serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
serv_adr.sin_port = htons(atoi(argv[2]));

connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
```

Example: uecho_con_client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 30
void error_handling(char *message);

int main(int argc, char *argv[])
{
   int sock;
   char message[BUF_SIZE];
   int str_len;
   socklen_t adr_sz;

   struct sockaddr_in serv_adr, from_adr;
   if (argc != 3) {
      printf("Usage : %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   sock = socket(PF_INET, SOCK_DGRAM, 0);
   if (sock == -1)
      error_handling("socket() error");
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
   serv_adr.sin_port = htons(atoi(argv[2]));

   connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

   while(1)
   {
      fputs("Insert message(q to quit): ", stdout);
      fgets(message, sizeof(message), stdin);
      if (!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
         break;

      write(sock, message, strlen(message));

      str_len = read(sock, message, sizeof(message)-1);
      message[str_len] = 0;

      printf("Message from server: %s", message);
   }
   close(sock);
   return 0;
}
```

connect()

write()

read()

Network Programming (Day 2): roadmap

- UDP Socket Programming
- Socket Options
- Multicast and Broadcast
- Multi-Process Programming #1
- Multi-Process Programming #2

Socket Options

- To modify some socket properties.

- Three ways to get and set the socket options
  - getsockopt(), setsockopt()
  - fcntl(): nonblocking I/O
  - Iocntl()

- System dependent

getsockopt() & setsockopt()

```
#include <sys/type.h>
#include <sys/socket.h>
int getsockopt(int sock, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int sock, int level, int optname, cont void *optval, socklen_t *optlen);
```

Get and set options on sockets

- *sock*: socket file descriptor
- *level*: protocol level at which the option resides
- *optname*: name of option
- *optval*: pointer to a variable from which the
- *optlen*: size of the option variable

- Return value
  - Success: 0
  - Error: -1
- Example

```
state = getsockopt(tcp_sock, SOL_SOCKET, SO_TYPE,
                        (void*)&sock_type, &optlen);
if (state)
   error_handling("getsockopt() error!");
printf("Socket type one: %d \n", sock_type);
```

Socket Options

Protocol Level	Option Name	Get	Set
SOL_SOCKET	SO_SNDBUF	O	O
	SO_RCVBUF	O	O
	SO_REUSEADDR	O	O
	SO_KEEPALIVE	O	O
	SO_BROADCAST	O	O
	SO_DONTROUTE	O	O
	SO_OOBINLINE	O	O
	SO_ERROR	O	X
	SO_TYPE	O	X
IPPROTO_IP	IP_TOS	O	O
	IP_TTL	O	O
	IP_MULTICAST_TTL	O	O
	IP_MULTICAST_LOOP	O	O
	IP_MULTICAST_IF	O	O
IPPROTO_TCP	TCP_KEEPALIVE	O	O
	TCP_NODELAY	O	O
	TCP_MAXSEG	O	O

SOL_SOCKET: SO_TYPE

- SO_TYPE: get the socket type (e.g., SOCK_STREAM, SOCK_DGRAM)
  - Example: sock_type.c

```
int tcp_sock, udp_sock;
int sock_type;
socklen_t optlen;
int state;

optlen = sizeof(sock_type);
tcp_sock = socket(PF_INET, SOCK_STREAM, 0);
udp_sock = socket(PF_INET, SOCK_DGRAM, 0);
printf("SOCK_STREAM: %d \n", SOCK_STREAM);
printf("SOCK_DGRAM: %d \n", SOCK_DGRAM);

state = getsockopt(tcp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
printf("Socket type one: %d \n", sock_type);

state = getsockopt(udp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
printf("Socket type two: %d \n", sock_type);
```

SOL_SOCKET: SO_RCVBUF, SO_SNDBUF

- SO_RCVBUF: gets or sets the maximum socket receive buffer in bytes
- SO_SNDBUF: gets or sets the maximum socket send buffer in bytes
  - Example: set_buf.c

```
int sock;
int snd_buf = 1024*3, rcv_buf = 1024*3;
int state;
socklen_t len;

sock = socket(PF_INET, SOCK_STREAM, 0);
state = setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, sizeof(rcv_buf));
state = setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, sizeof(snd_buf));

len = sizeof(snd_buf);
state = getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);

len = sizeof(rcv_buf);
state = getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);

printf("Input buffer size: %d \n", rcv_buf);
printf("Output buffer size: %d \n", snd_buf);
```

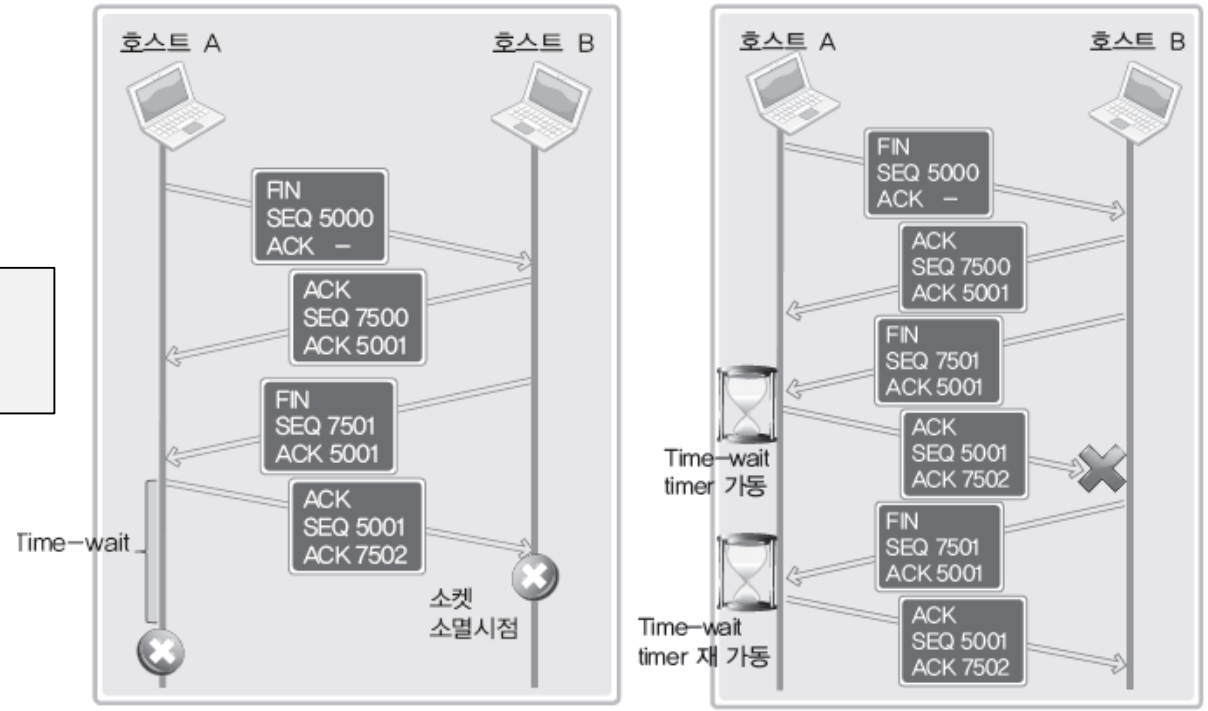
The actual value set in the kernel may not be exactly the same as the value set by the user. It is set by the kernel to twice the size of the set value.

SOL_SOCKET: SO_REUSEADDR

- SO_REUSEADDR: allows a listening server to start and bind the port even if previously established connection use that port
  - Cases to need SO_REUSEADDR
    - A server in the state of TIME_WAIT
    - A multi-home server

```
optlen = sizeof(option);
option = TRUE;
setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, &option, optlen);
```

<TIME_WAIT>



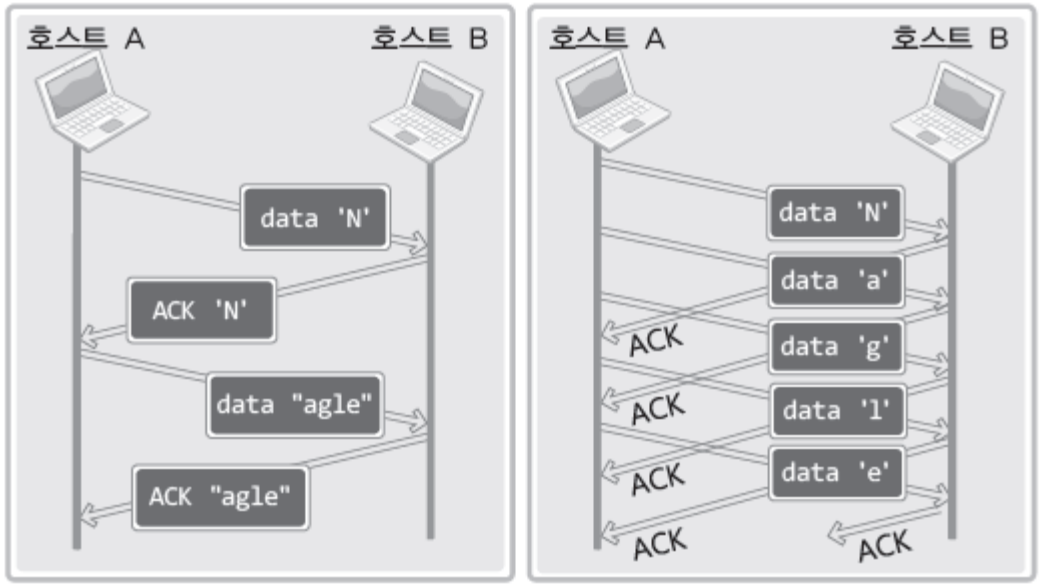
IPPROTO_TCP: TCP_KEEPALIVE, TCP_MAXSEG

- TCP_KEEPALIVE
  - It specifies the idle time in second for the connection before TCP starts sending keepalive probe.
    - Default 2 hours.
  - This option is effective only when the SO_KEEPALIVE socket option enabled.

- TCP_MAXSEG
  - It allows to get or set the maximum segment size (MSS) for TCP connection.

IPPROTO_TCP: TCP_NODELAY

- TCP_NODELAY
  - If set, disable the TCP's Nagle algorithm.
    - Default: this algorithm enabled
  - Nagle algorithm
    - When data come from application, send the first byte. Then
    - 1) Buffer all the rest until the outstanding byte is ACKed.
    - 2) Send a new packet if data fill the half the window or a maximum segment
      - Better to be disabled if used on mouse movements.



Nagle 알고리즘 ON　　Nagle 알고리즘 OFF

IPPROTO_TCP: TCP_NODELAY

- TCP_NODELAY
  - Set TCP_NODELAY (= disable the Nagle algorithm)

```
int opt_val = 1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void*)&opt_val, sizeof(opt_val));
```

  - Get option for TCP_NODELAY

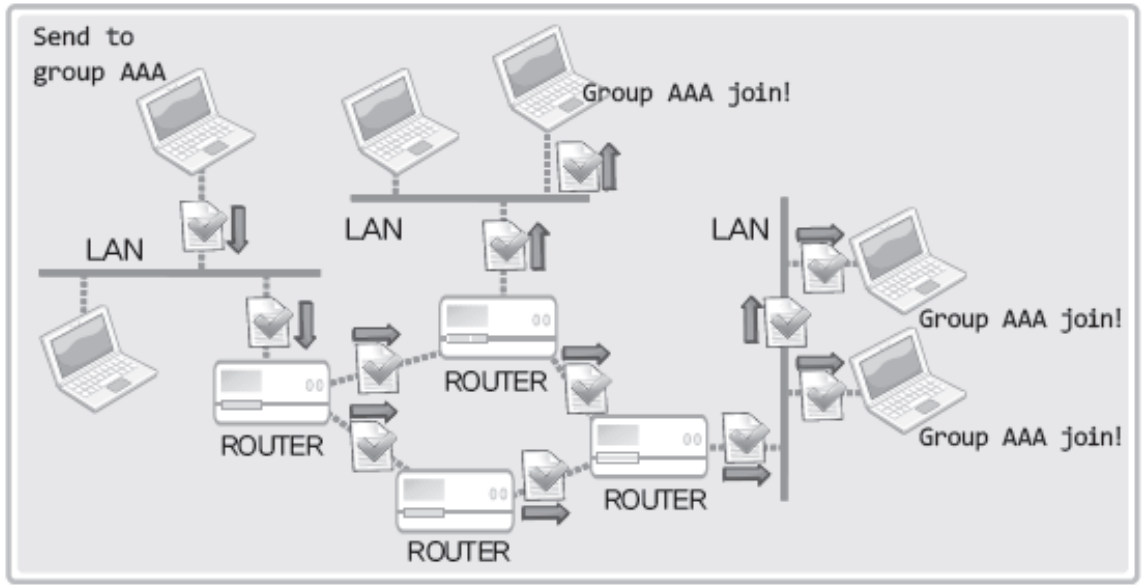
```
int opt_val;
socklen_t opt_len;
opt_len = sizeof(opt_val);
getsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void*)&opt_val, &opt_len);
```

Network Programming (Day 2): roadmap

- UDP Socket Programming
- Socket Options
- Multicast and Broadcast
- Multi-Process Programming #1
- Multi-Process Programming #2

Multicast

- Server transmits data once for the specific multicast group.
- All clients join in the multicast group can receive all data.
- The client needs to join the multicast group to receive data.
- Based on UDP
- Multicast IP addresses (Class D)
  - 224.0.0.0 ~ 239.255.255.255

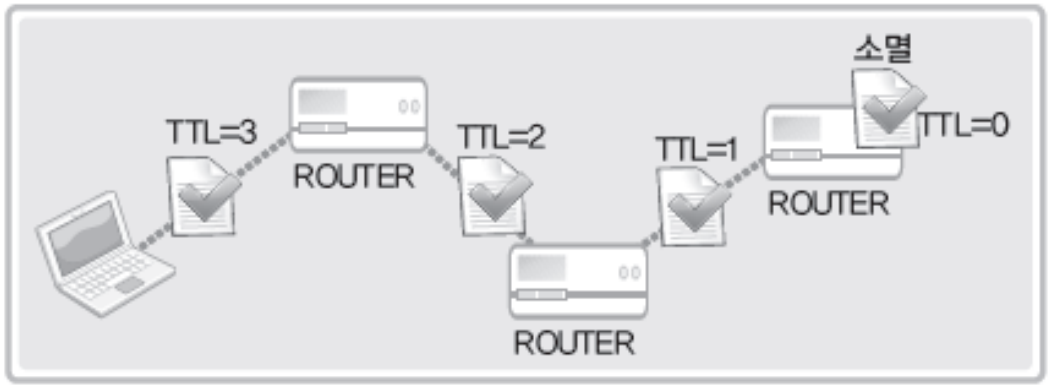


Global range from 224.0.0.0 to 239.255.255.255
- Reserved: 224.0.0.0 ~ 224.0.0.255
- Internet-wide addresses: 224.0.1.0 ~ 238.255.255.255
- Local addresses: 239.0.0.0 ~ 239.255.255.255

Some special addresses
- 224.0.0.1: all multicast systems on a subnet
- 224.0.0.2: all multicast routers on a subnet

Programming for Multicast

- Sender
  - Make a UDP socket
  - Configure a TTL (Time to Live)
    - Default value = 1
  - Send a data to multicast address



- Receiver
  - Make a UDP socket
  - Join a multicast group

Multicast Socket Options

- IP_ADD_MEMBERSHIP
  - Join a multicast group
  - Related structure
- IP_DROP_MEMBERSHIP
  - Leave a multicast membership

- Related structure

```
struct ip_mreq {
   struct in_addr imr_multiaddr;
   struct in_addr imr_interface;
};
```

Multicast Socket Options

- IP_MULTICAST_IF
  - Specify default interface for outgoing multicasts
  - Specified as an IN_ADDR for IPv4
- IP_MULTICAST_LOOP
  - Enable or disable loopback of outgoing multicasts
- IP_MULTICAST_TTL
  - Specify TTL for multicasts
  - If no specified, default value is 1.

Example: news_sender.c

- Sender

```
int send_sock;
struct sockaddr_in mul_adr;
int time_live = TTL;
FILE *fp;
char buf[BUF_SIZE];

if (argc != 3) {
   printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
   exit(1);
}

send_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&mul_adr, 0, sizeof(mul_adr));
mul_adr.sin_family = AF_INET;
mul_adr.sin_addr.s_addr = inet_addr(argv[1]);  // Multicast IP
mul_adr.sin_port = htons(atoi(argv[2]));          // Multicast Port
```

```
setsockopt(send_sock, IPPROTO_IP, IP_MULTICAST_TTL, (void*)&time_live,
                sizeof(time_live));

if ((fp = fopen("news.txt", "r")) == NULL)
   error_handling("fopen() error");

while(!feof(fp))   /* Broadcasting */
{
   fgets(buf, BUF_SIZE, fp);
   sendto(send_sock, buf, strlen(buf), 0, (struct sockaddr*)&mul_adr,
             sizeof(mul_adr));
   sleep(2);
}
close(send_sock);
```

Example: news_receiver.c

- Sender

```
int recv_sock;
int str_len;
char buf[BUF_SIZE];
struct sockaddr_in adr;
struct ip_mreq join_adr;

if (argc != 3) {
   printf("Usage : %s <GroupIP> <PORT>\n", argv[0]);
   exit(1);
}

recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_addr.s_addr = htonl(INADDR_ANY);
adr.sin_port = htons(atoi(argv[2]));
```

```
if (bind(recv_sock, (struct sockaddr*) &adr, sizeof(adr))==-1)
   error_handling("bind() error");

join_adr.imr_multiaddr.s_addr = inet_addr(argv[1]);     // Multicast IP
join_adr.imr_interface.s_addr = htonl(INADDR_ANY);

setsockopt(recv_sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                (void*)&join_adr, sizeof(join_adr));

while(1)
{
   str_len = recvfrom(recv_sock, buf, BUF_SIZE-1, 0, NULL, 0);
   if (str_len<0)
      break;
   buf[str_len]=0;
   fputs(buf, stdout);
}
close(recv_sock);
```

Broadcast

- Destination is the all of hosts reside in same network
- Based on UDP
- Broadcast IP addresses
  - Directed broadcast
    - Host ID part = all 1’s
  - Limited broadcast (same subnet)
    - 255.255.255.255

Programming for Broadcast

- Almost same with the UDP socket programming except broadcast IP address and socket option

- Socket option: SO_BROADCAST
  - SOL_SOCKET level options
  - Enable or disable the ability of the process to send broadcast message. (only datagram socket)

Example: news_sender_brd.c

- Sender

```
int send_sock;
struct sockaddr_in broad_adr;
FILE *fp;
char buf[BUF_SIZE];
int so_brd = 1;

if (argc != 3) {
   printf("Usage : %s <Boradcast IP> <PORT>\n", argv[0]);
   exit(1);
}

send_sock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&broad_adr, 0, sizeof(broad_adr));
broad_adr.sin_family = AF_INET;
broad_adr.sin_addr.s_addr = inet_addr(argv[1]);
broad_adr.sin_port = htons(atoi(argv[2]));
```

```
setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST,
            (void*)&so_brd, sizeof(so_brd));
if ((fp = fopen("news.txt", "r")) == NULL)
   error_handling("fopen() error");

while (!feof(fp))
{
   fgets(buf, BUF_SIZE, fp);
   sendto(send_sock, buf, strlen(buf),
          0, (struct sockaddr*)&broad_adr, sizeof(broad_adr));
   sleep(2);
}

close(send_sock);
```

Example: news_receiver_brd.c

- Sender

```
int recv_sock;
struct sockaddr_in adr;
int str_len;
char buf[BUF_SIZE];

If (argc!=2) {
   printf("Usage : %s  <PORT>\n", argv[0]);
   exit(1);
}

recv_sock = socket(PF_INET, SOCK_DGRAM, 0);

memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_addr.s_addr = htonl(INADDR_ANY);
adr.sin_port = htons(atoi(argv[1]));
```

```
if (bind(recv_sock, (struct sockaddr*)&adr, sizeof(adr)) == -1)
   error_handling("bind() error");

while (1)
{
   str_len = recvfrom(recv_sock, buf, BUF_SIZE-1, 0, NULL, 0);
   if (str_len<0)
      break;
   buf[str_len]=0;
   fputs(buf, stdout);
}

close(recv_sock);
return 0;
```

Network Programming (Day 2): roadmap

- UDP Socket Programming
- Socket Options
- Multicast and Broadcast
- Multi-Process Programming #1
- Multi-Process Programming #2

Multiple access server

- Multiple access server
  - It allows multiple connections for the multiple clients and provides the service simultaneously.

- Multi-process based server
  - Create multiple processes for the multiple clients
- Multiplexing based sever
  - Grouping the I/O clients for the multiple clients
- Multi-thread based server
  - Create multiple threads for the multiple clients

Process

- Process
  - Running program
  - Includes memory and resources for the running program
  - Multi process operating system can create two more processes.
  - Operating system assigns a process ID (PID) for each created process

```
pi@raspberrypi:~/netprog $ ps au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       513  0.0  0.2   5620  2744 tty1     Ss   00:36   0:00 /bin/login -f
root       514  0.0  6.2 280860 59500 tty7     Ssl+ 00:36   0:11 /usr/lib/xorg/Xorg :0 -seat sea
pi         630  0.0  0.3   8492  3700 tty1     S+   00:36   0:00 -bash
pi        1140  0.0  0.4   8492  4000 pts/0    Ss   00:39   0:01 -bash
pi       12480  0.0  0.2   9788  2528 pts/0    R+   04:23   0:00 ps au
```

Create a new process: fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Create a child process

- fork() creates a new process by duplicating the calling process.
- The new process is referred to as the child process.
- The calling process is referred to as the parent process.
- The child process and the parent process run in separate memory spaces.
- At the time of fork() both memory spaces have the same content.
- Memory writes, file mappings, and unmappings performed by one of the processes do not affect the other.
- Return value
  - Success on child process: 0
  - Success on parent process: child PID
  - Error: -1

Example: fork.c

Parent Process

```
int gval = 10;
int main(void)
{
   int lval = 20;
   pid_t pid;
   lval += 5;
   gval++;
   pid = fork();    // pid is child's PID
   if (pid == 0)
      gval++;
   else
      lval++;
   ......
}
```

Copy

Child Process

```
int gval = 10;
int main(void)
{
   int lval = 20;
   pid_t pid;
   lval += 5;
   gval++;
   pid = fork();  // pid is 0
   if (pid == 0)
      gval++;
   else
      lval++;
   ......
}
```

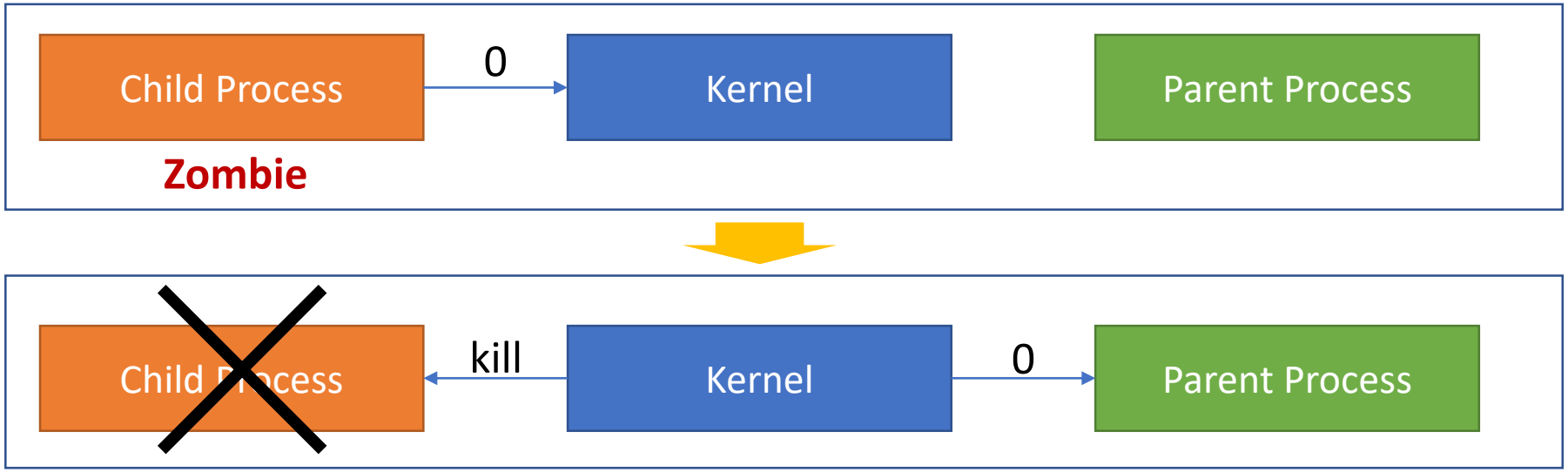
When fork(),
gval is 11 and lval is 25.

Result

```
[gym@localhost practice06]$ ./fork
Parent Proc: [11, 26]
Child Proc: [12, 25]
```

Zombie Process

- Zombie state
  - Maintains information about the child process for the parent to fetch at some later time → Resource is still remained
- After a child process is terminated (call exit() or return 0 in main), the child process will be in a zombie state until its parent process handles SIGCHLD signal from its child process.



Example: zombie.c

- Check zombie process

```
pid_t pid = fork();
if (pid == 0)    // if child process
{
   puts("Hi, I am a child process");
}
else
{
   printf("Child process ID: %d \n", pid);
   sleep(30);    // Sleep 30 sec.
}
if (pid == 0)
   puts("End child process");
else
   puts("End parent process");
Return 0;
```

Result

```
[gym@localhost practice06]$ ./zombie
Child Process ID: 17073
Hi I'am a child process
End child process
```

```
[gym@localhost ~]$ ps a
   PID TTY      STAT   TIME COMMAND
  2030 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  9687 pts/1    Ss     0:00 -bash
  9715 pts/1    S+     0:00 bash
 15508 pts/10   Ss     0:00 /usr/bin/bash --init-file /home/gym/.vsc
 16923 pts/0    Ss     0:00 /usr/bin/bash --init-file /home/gym/.vsc
 17072 pts/10   S+     0:00 ./zombie
 17073 pts/10   Z+     0:00 [zombie] <defunct>
 17091 pts/0    R+     0:00 ps a
```

Handling zombie process: wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

Wait for process to change state

- Suspends the calling process until one of the children terminates (block function)
- *wstatus*: If status is not NULL, waitpid() store status information in the int to which it points. This integer can be inspected with the macros
- Return value
  - Success: child process ID
  - Error: -1
- wait() is equivalent to waitpid(-1, &wstatus, 0);

Handling zombie process: waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Wait for process to change state

- Suspends execution of the calling process until a child specified by pid argument has changed state.
- *pid*: a child process ID that its parent process waits to stop
  - If pid is -1, the parent process waits for any child process to stop.
- *wstatus*: same with the *status* in wait()
- *options*
  - By default, waitpid() waits only for terminated children, but this behavior is modifiable via the options argument.
  - WNOHANG: return immediately if no child has exited
- Return value
  - Success: child process ID
  - Error: -1

Handling zombie process: returning status

- Macro function for handling returning status value from wait() and waitpid()

Macro function	Return value
WIFEXITED(wstatus)	returns true if the child terminated normally
WEXITSTATUS(wstatus)	returns the least significant 8 bits of the exit status that the child specified in if WIFEXITED returned true.
WIFSIGNALED(wstatus)	returns true if the child process was terminated by a signal (abnormal termination).
WTERMSIG(wstatus)	returns the number of the signal that caused the child process to terminate. This macro should be employed only if WIFSIGNALED returned true.

Example: wait.c

```
int main(int argc, char *argv[])
{
   int status;
   pid_t pid = fork();

   if (pid == 0)
   {
      return 3;
   }
   else
   {
      printf("Child PID: %d \n", pid);
      pid = fork();
      if (pid == 0)
      {
         exit(7);
      }
      else
      {
         printf("Child PID: %d \n", pid);
         wait(&status);
         if (WIFEXITED(status))
            printf("Child send one: %d \n", WEXITSTATUS(status));

         wait(&status);
         if (WIFEXITED(status))
            printf("Child send two: %d \n", WEXITSTATUS(status));
         sleep(10);     // Sleep 10 sec.
      }
   }
   return 0;
}
```

Result

```
[gym@localhost practice06]$ ./wait
Child PID: 17820
Child PID: 17821
Child send one: 3
Child send two: 7
```

Example: waitpid.c

```
int main(int argc, char *argv[])
{
   int status;
   pid_t pid = fork();

   if (pid ==0 )
   {
      sleep(15);
      return 24;
   }
   else
   {
      while (!waitpid(-1, &status, WNOHANG))      // wait()
      {
         sleep(3);
         puts("sleep 3sec.");
      }
      if (WIFEXITED(status))
         printf("Child send %d \n", WEXITSTATUS(status));
   }
   return 0;
}
```

Result

```
[gym@localhost practice06]$ ./waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
```

Network Programming (Day 2): roadmap

- UDP Socket Programming
- Socket Options
- Multicast and Broadcast
- Multi-Process Programming #1
- Multi-Process Programming #2

Signals

- Signal
  - A signal is a notification to a process that an event has occurred.
  - Signals are sometimes called “software interrupts”

- Signals can be sent
  - By one process to another process (or itself)
  - By the kernel to a process
  - Ex. SIGCHLD signal
    - A signal sent to the parent of the terminating process by the kernel whenever a process terminates

Signals

- Signal disposition (action associated with a signal)
  - Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.
  - We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler
  - We can ignore a signal by setting its disposition to SIG_IGN.
  - We can set default disposition for a signal by setting its disposition to SIG_DFL.

Signals

- Signal types

Signal name	Description	Default action
SIGALRM	Time-out occurs (alarm)	Terminate
SIGCHLD	Change in status of a child	Ignore
SIGINT	Terminal interrupt character (ctrl+C)	Terminate
SIGKILL	Termination	Terminate
SIGPIPE	Write to pipe with no readers	Terminate
SIGSTOP	Stop (ctrl+S)	Stop process

signal()

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Define the action associated with a signal

- sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler")
- *signum*: the name of signal
- *func*: the address of a function to be called when the signal occurs, SIG_IGN or SIG_DFL

- Return value
  - Success: the previous value of the signal handler
  - Error: SIG_ERR

Example: signal.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void timeout(int sig)
{
   if (sig == SIGALRM)
      puts("Time out!");

   alarm(2);
}
void keycontrol(int sig)
{
   if (sig == SIGINT)
      puts("CTRL+C pressed");
}
```

```c
int main(int argc, char *argv[])
{
   int i;
   signal(SIGALRM, timeout);
   signal(SIGINT, keycontrol);
   alarm(2);

   for (i = 0; i < 3; i++)
   {
      puts("wait...");
      sleep(10);
   }
   return 0;
}
```

Result

```
[gym@localhost practice07]$ ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

```
[gym@localhost practice07]$ ./signal
wait...
^CCTRL+C pressed
wait...
Time out!
wait...
Time out!
```

sigaction()

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Examine and change a signal action

- used to change the action taken by a process on receipt of a specific signal.
- *signum*: the name of signal
- *act*: the new action
- *oldact*: the previous actions
- Return value
  - Success: 0
  - Error: -1

- struct sigaction

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

  - sa_handler: the address of a function to be called when the signal occurs
  - sa_mask: specifies a set of signals that will be blocked when the signal handler is called
  - sa_flags: options. Basically 0

Example: sigaction.c

```
void timeout(int sig)
{
   if (sig == SIGALRM)
      puts("Time out!");
   alarm(2);
}

int main(int argc, char *argv[])
{
   int i;
   struct sigaction act;
   act.sa_handler = timeout;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   sigaction(SIGALRM, &act, 0);

   alarm(2);

   for (i = 0; i < 3; i++)
   {
      puts("wait...");
      sleep(10);
   }
   return 0;
}
```

Result

```
[gym@localhost practice07]$ ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

Example: remove_zombie.c

```c
void read_childproc(int sig)
{
   int status;
   pid_t id = waitpid(-1, &status, WNOHANG);
   if (WIFEXITED(status))
   {
      printf("Removed proc id: %d \n", id);
      printf("Child send: %d \n", WEXITSTATUS(status));
   }
}

int main(int argc, char *argv[])
{
   pid_t pid;
   struct sigaction act;
   act.sa_handler = read_childproc;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   sigaction(SIGCHLD, &act, 0);

   pid = fork();
   if (pid == 0)
   {
      puts("Hi! I'm child process1");
      sleep(10);
      return 12;
   }
   else
   {
      printf("Child proc id1: %d \n", pid);
      pid = fork();
      if (pid == 0)
      {
         puts("Hi! I'm child process2");
         sleep(10);
         exit(24);
      }
      else
      {
         int i;
         printf("Child proc id2: %d \n", pid);
         for (i = 0; i < 5; i++)
         {
            puts("wait...");
            sleep(5);
         }
      }
   }
   return 0;
}
```

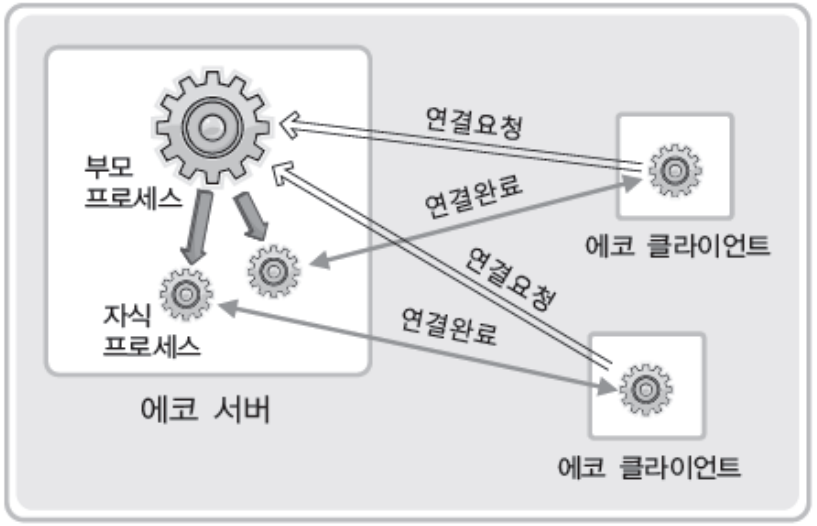
Result

```
[gym@localhost practice07]$ ./remove_zombie
Child proc id1: 19801
Hi! I'm child process1
Child proc id2: 19802
wait...
Hi! I'm child process2
wait...
Removed proc id: 19802
Child send: 24
wait...
Removed proc id: 19801
Child send: 12
wait...
wait...
```

```
[gym@localhost ~]$ ps a
  PID TTY      STAT   TIME COMMAND
 2030 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
 9687 pts/1    Ss     0:00 -bash
 9715 pts/1    S+     0:00 bash
15508 pts/10   Ss     0:00 /usr/bin/bash --init-file /home/gym/.
16923 pts/0    Ss     0:00 /usr/bin/bash --init-file /home/gym/.
19725 pts/3    Ss+    0:00 /usr/bin/bash --init-file /home/s2180
19800 pts/10   S+     0:00 ./remove_zombie
19801 pts/10   S+     0:00 ./remove_zombie
19802 pts/10   S+     0:00 ./remove_zombie
19842 pts/0    R+     0:00 ps a
```

Concurrent server

- Concurrent server
  - A separate process to handle each client
  - The main server process creates a new service process to handle each client
  - Most TCP servers are concurrent: for ease of connection management



- 1단계 에코 서버(부모 프로세스)는 accept 함수호출을 통해서 연결요청을 수락한다.
- 2단계 이때 얻게 되는 소켓의 파일 디스크립터를 자식 프로세스를 생성해서 넘겨준다.
- 3단계 자식 프로세스는 전달받은 파일 디스크립터를 바탕으로 서비스를 제공한다.

Iterative vs. Concurrent servers

Iterative server skeleton

```
int sockfd, newsockfd;
if ((sockfd = socket(…)) < 0)
   err_sys(“socket error”);
if ((bind(sockfd, …) < 0)
   err_sys(“bind error”);
if (listen(sockfd, 5))
   err_sys(“listen error”);
for (;;) {
   newsockfd = accept(sockfd, …);
   if (newsockfd < 0)
      err_sys(“accept error”);
   doit(newsockfd);
   close(newsockfd);
}
```

Concurrent server skeleton

```
int sockfd, newsockfd;
if ((sockfd = socket(…)) < 0)
   err_sys(“socket error”);
if ((bind(sockfd, …) < 0)
   err_sys(“bind error”);
if (listen(sockfd, 5))
   err_sys(“listen error”);
for (;;) {
   newsockfd = accept(sockfd, …);
   if (newsockfd < 0)
      err_sys(“accept error”);
   if (fork() == 0) {
      close(sockfd);
      doit(newsockfd);
      exit(0);
   }
   else {
      close(newsockfd);
   }
}
```

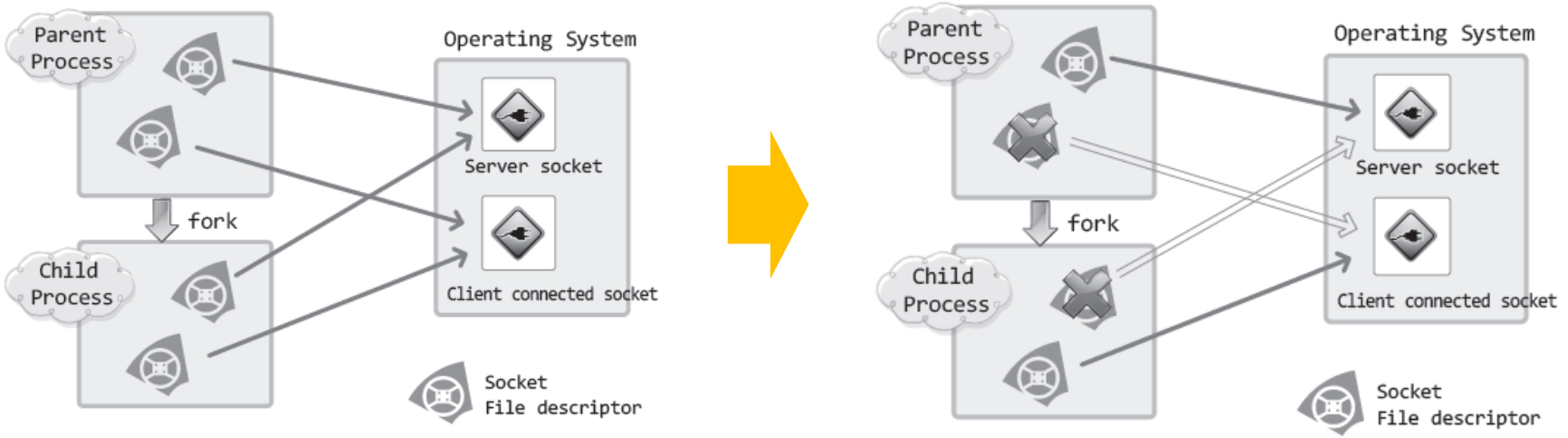
Example: echo_mpserv.c

```
int main(int argc, char *argv[])
{
   int serv_sock, clnt_sock;
   struct sockaddr_in serv_adr, clnt_adr;

   pid_t pid;
   struct sigaction act;
   socklen_t adr_sz;
   int str_len, state;
   char buf[BUF_SIZE];
   if (argc != 2) {
      printf("Usage : %s <port>\n", argv[0]);
      exit(1);
   }

   act.sa_handler = read_childproc;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   state = sigaction(SIGCHLD, &act, 0);
   serv_sock = socket(PF_INET, SOCK_STREAM, 0);
   memset(&serv_adr, 0, sizeof(serv_adr));
   serv_adr.sin_family = AF_INET;
   serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
   serv_adr.sin_port = htons(atoi(argv[1]));

   if (bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr)) == -1)
      error_handling("bind() error");
   if (listen(serv_sock, 5) == -1)
      error_handling("listen() error");
   while(1)
   {
      adr_sz = sizeof(clnt_adr);
      clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
      if (clnt_sock == -1)
         continue;
      else
         puts("new client connected...");

      pid = fork();
      if (pid == -1)
      {
         close(clnt_sock);
         continue;
      }
      if (pid == 0)
      {
         close(serv_sock);
         while ((str_len = read(clnt_sock, buf, BUF_SIZE)) != 0)
            write(clnt_sock, buf, str_len);
         close(clnt_sock);
         puts("client disconnected...");
         return 0;
      }
      else
         close(clnt_sock);
   }
   close(serv_sock);
   return 0;
}
```

```
void read_childproc(int sig)
{
   pid_t pid;
   int status;
   pid = waitpid(-1, &status, WNOHANG);
   printf("removed proc id: %d \n", pid);
}
```

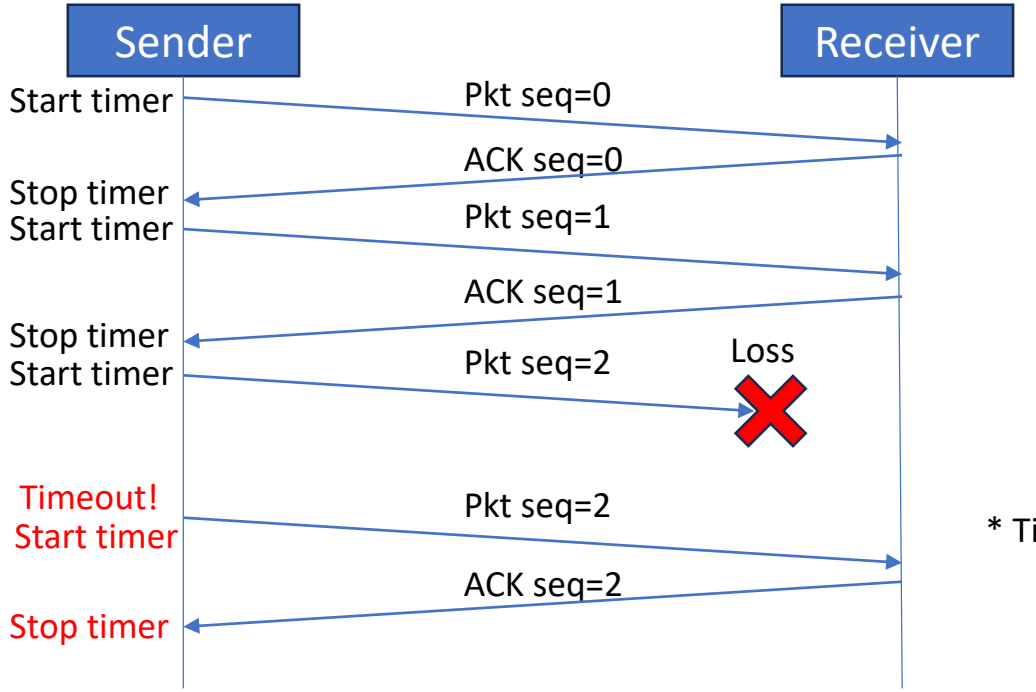
fork() and Socket descriptor

- When fork() is executed, a socket is not copied but shared by a parent and a child process. (the socket descriptor is only copied.)
  - The socket is handled by a kernel.
- In case that all the processes that share a socket close the socket, the kernel removes the socket.



숙제 #2

- Stop-and-Wait Protocol 구현
  - uecho_server.c 와 uecho_client.c는 신뢰성 있는 데이터 전송을 보장하지 않습니다. (손실된 패킷을 복구하는 기능이 없음)
  - 이를 보완하여 신뢰성 있는 데이터 전송을 보장하는 Stop-and-Wait Protocol 기반의 프로그램을 구현하세요.
  - 파일 전송이 완료되면 Throughput을 출력하세요. (Throughput = 받은 데이터 양 / 전송 시간)



* Timeout 시간은 어떻게 설정해야할까요?