Network Programming

Client Server Paradigm

- ***Server application is "listener"***
- Waits for incoming message
- Performs service
- Returns results
- ***Client application establishes connection***
- Sends message to server
- Waits for return message

Client-Server Communication

• Clients and servers exchange messages through transport protocols; e.g., TCP or UDP
• Both client and server must have same protocol stack and both interact with transport

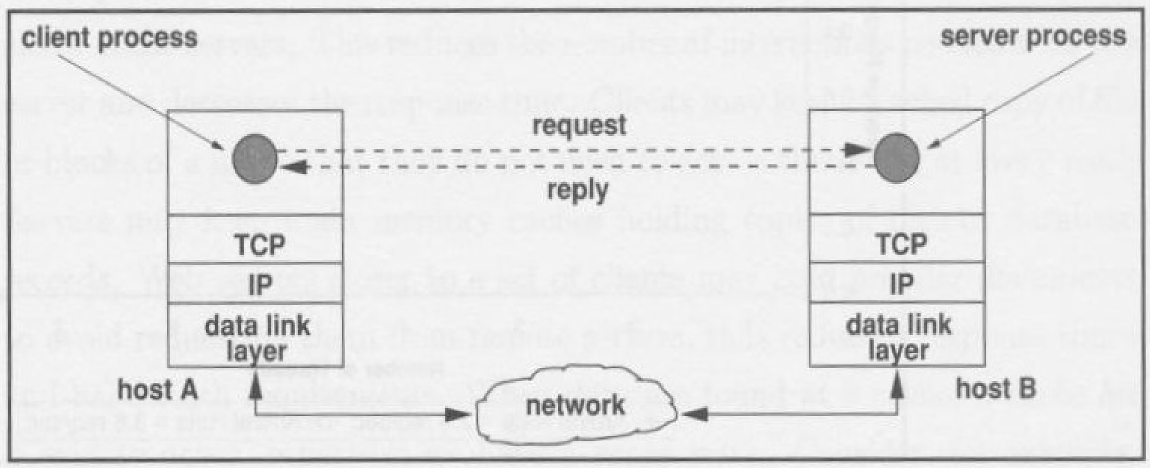
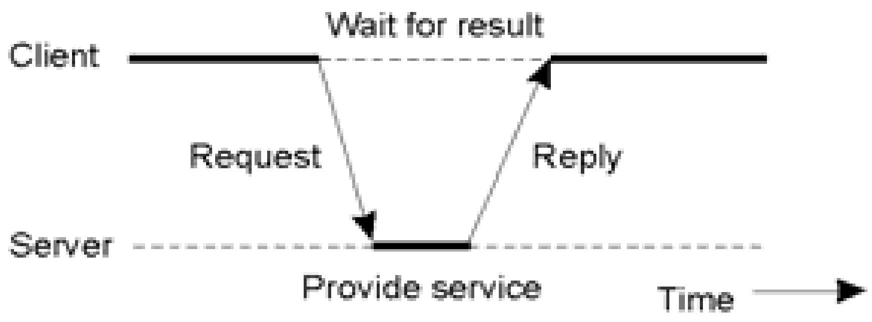
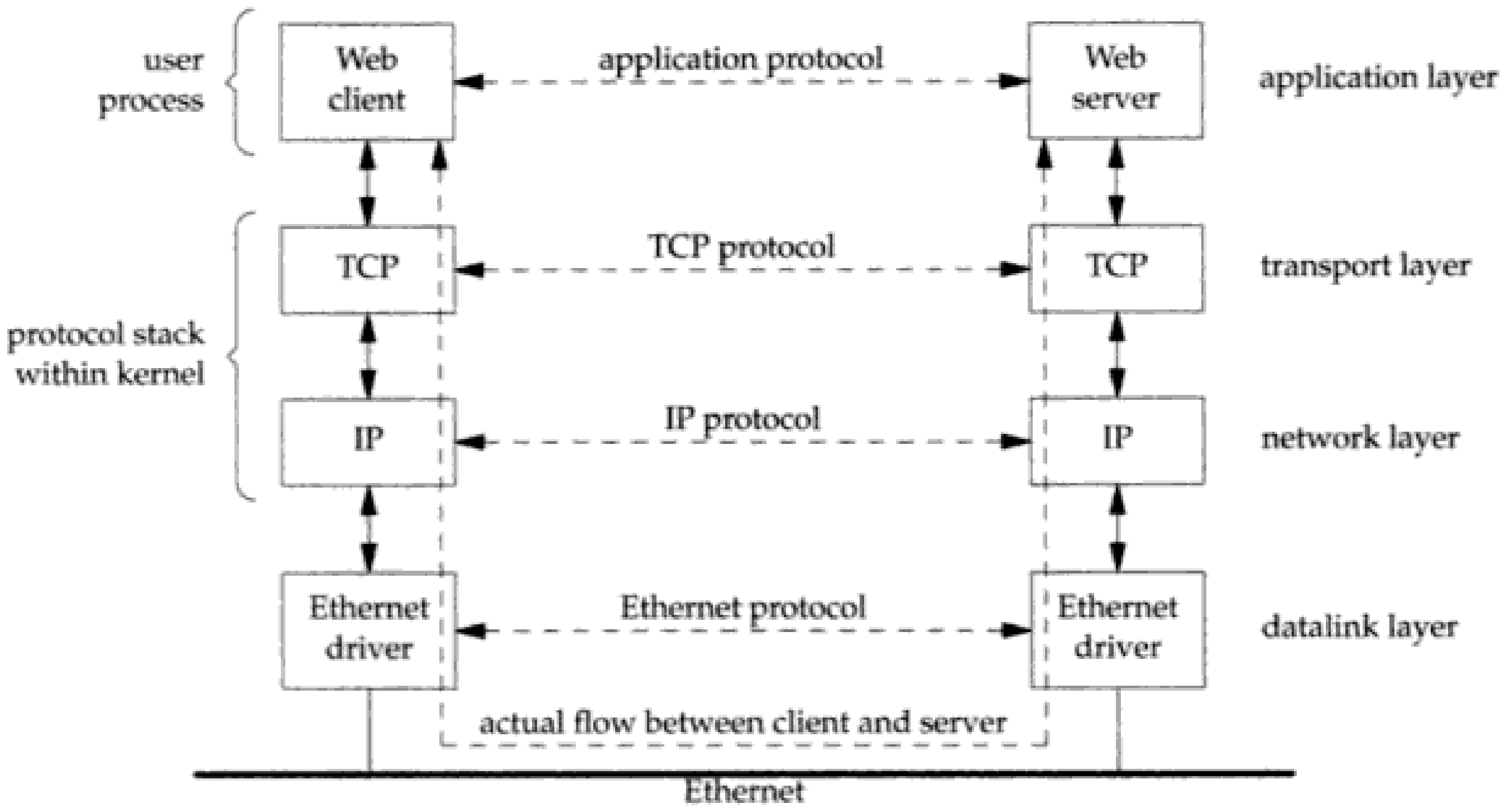


Figure 2.13. Client/server protocol: request/reply.

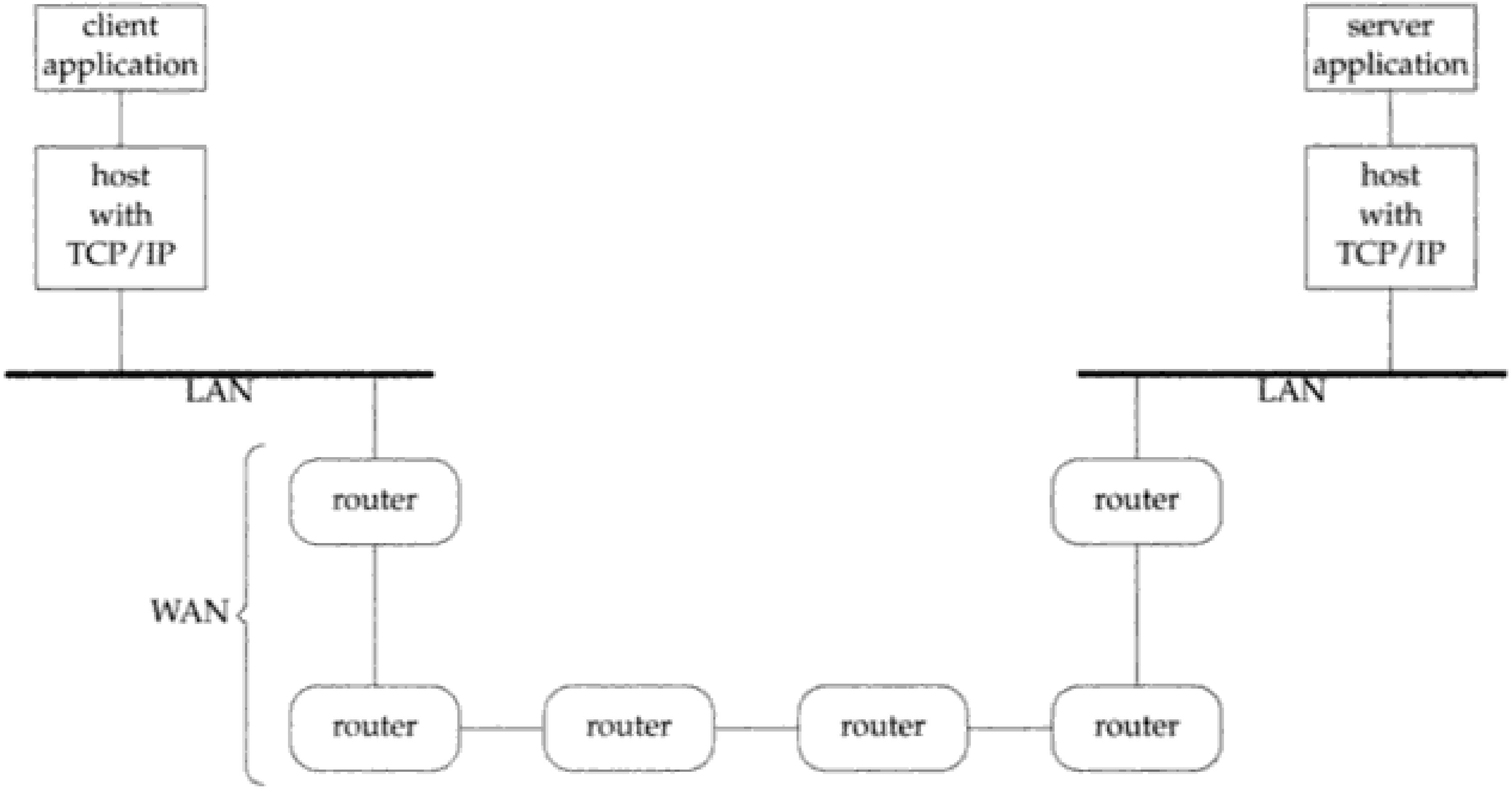
Interaction Between Client-Server



C S on same Ethernet



C S on different LAN



OSI Layers In Client Server

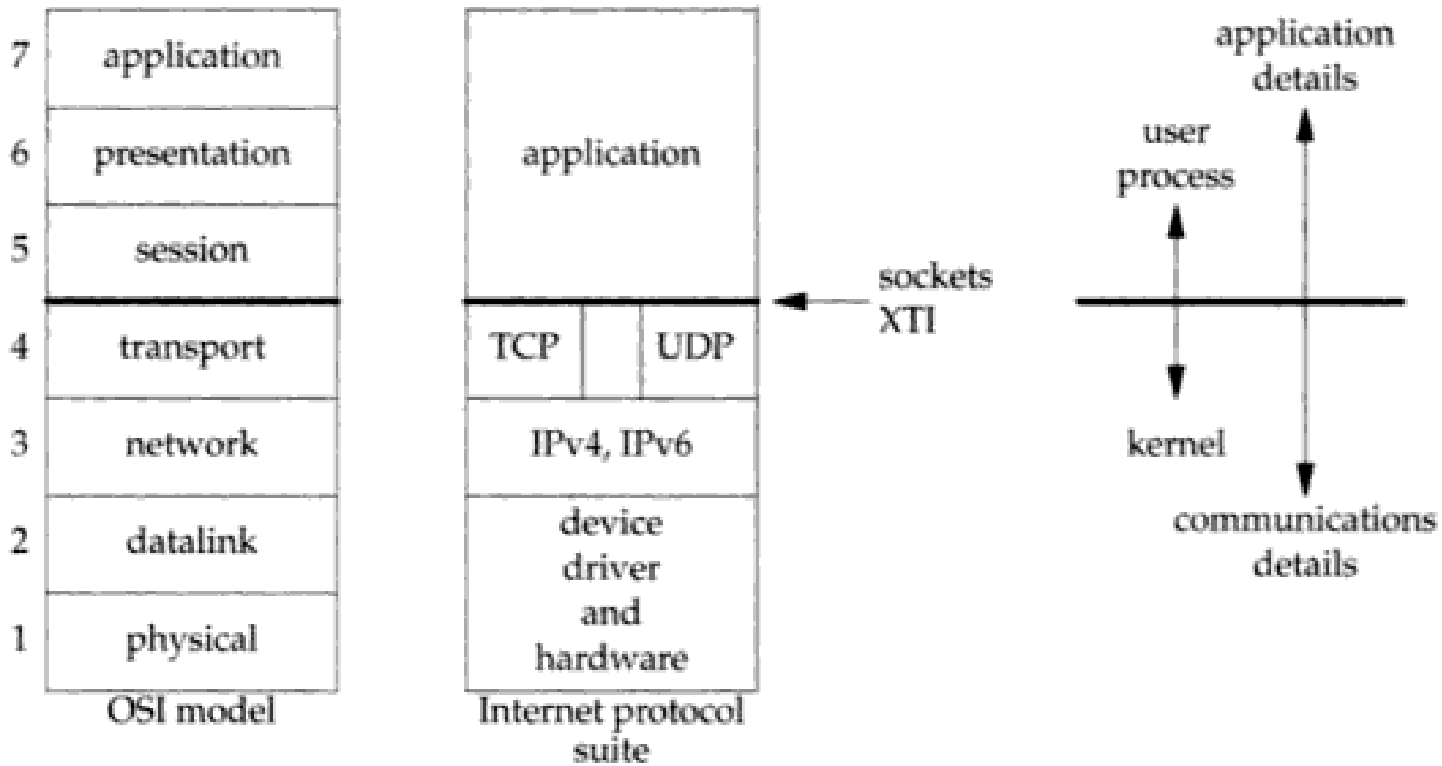


Figure 1.14 Layers in OSI model and Internet protocol suite.

Programming Interfaces: Sockets

Client-Server Communication Model

- Service Model
  - Concurrent:
    - Server processes multiple clients' requests simultaneously
  - Sequential:
    - Server processes only one client's requests at a time
  - Hybrid:
    - Server maintains multiple connections, but processes responses sequentially
- Client and server categories are not disjoint
  - A server can be a client of another server
  - A server can be a client of its own client

Sockets

```
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

- Creates a new socket
- SOCK_STREAM = TCP
- SOCK_DGRAM = UDP

Socket Address in General

Socket Addresses

- Need to be able to specify addresses
- Clients needs to
  - Specify the IP address of the destination
  - Specify the port number of the destination
  - (Obtain a local port number)
- Servers need to
  - Associate the (well-known) server port number to the socket
  - (Restrict the service to particular IP addresses)
- Addresses are specified in network byte order (which is big endian)

Sockaddr struct

IP Socket Addresses

```
struct sockaddr_in {

     sa_family_t       sin_family;     /* address family: AF_INET */
     u_int16_t         sin_port;       /* port in network byte order */
     struct in_addr    sin_addr;       /* internet address */
};

/* Internet address. */
struct in_addr {
     u_int32_t         s_addr;         /* address in network byte order */
};
```

- Small variations between different operating systems
- `inet_addr()` can be used to convert a dotted-quad string to a binary representation
  - in network byte order

```
in_addr_t inet_addr(const char *cp);
```

Socket Address

Requesting Socket Addresses

```
int  getsockopt(int  s, int level, int optname, void *optval,
socklen_t *optlen);

int setsockopt(int s, int  level,  int  optname,  const  void
*optval, socklen_t optlen);
```

- Getpeername
  - Get name (socket address) of the peer connected to the socket
  - Remote side
- Getsockname
  - Get name (socket address) of the socket

SOCKET

Sockets Are for Networks in General...

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

Domain (family)	*Purpose*	*Man page*
PF_UNIX, PF_LOCAL	Local communication	unix(7)
PF_INET	IPv4 Internet protocols	ip(7)
PF_INET6	IPv6 Internet protocols	
PF_IPX	IPX - Novell protocols	
PF_NETLINK	Kernel user interface device	netlink(7)
PF_X25	ITU-T X.25 / ISO-8208 protocol	x25(7)
PF_AX25	Amateur radio AX.25 protocol	
PF_ATMPVC	Access to raw ATM PVCs	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Low level packet interface	packet(7)

Type	*Purpose*
SOCK_STREAM	Sequenced, reliable, two-way, connection-based byte streams.
SOCK_DGRAM	Datagrams.
SOCK_SEQPACKET	Sequenced, reliable, two-way connection-based data transmission path for datagrams
SOCK_RAW	Raw network protocol access.
SOCK_RDM	A reliable datagram layer that does not guarantee ordering.
SOCK_PACKET	Obsolete and should not be used in new programs

Protocol	*Purpose*
IPPROTO_TCP	TCP (Only legal protocol for PF_INET + SOCK_STREAM)
IPPROTO_UDP	UDP (Only legal protocol for PF_INET + SOCK_DGRAM)
0	Default protocol for given domain and type

Socket Options

Socket Options

SOL_SOCKET	*Purpose*
SO_SNDBUF	Maximum send buffer size
SO_RCVBUF	Maximum receive buffer size
SO_KEEPALIVE	Enable/disable keep-alive messages (connection-oriented sockets)
SO_BROADCAST	Allow sending/reception of broadcast packets (datagram sockets)
SO_REUSEADDR	Allow reuse of local address in bind calls

SOL_IP	*Purpose*
IP_TTL	TTL field for packets sent
IP_ADD_MEMBERSHIP	Join a multicast group
IP_DROP_MEMBERSHIP	Leave a multicast
IP_MTU	Get current path MTU (connected sockets)

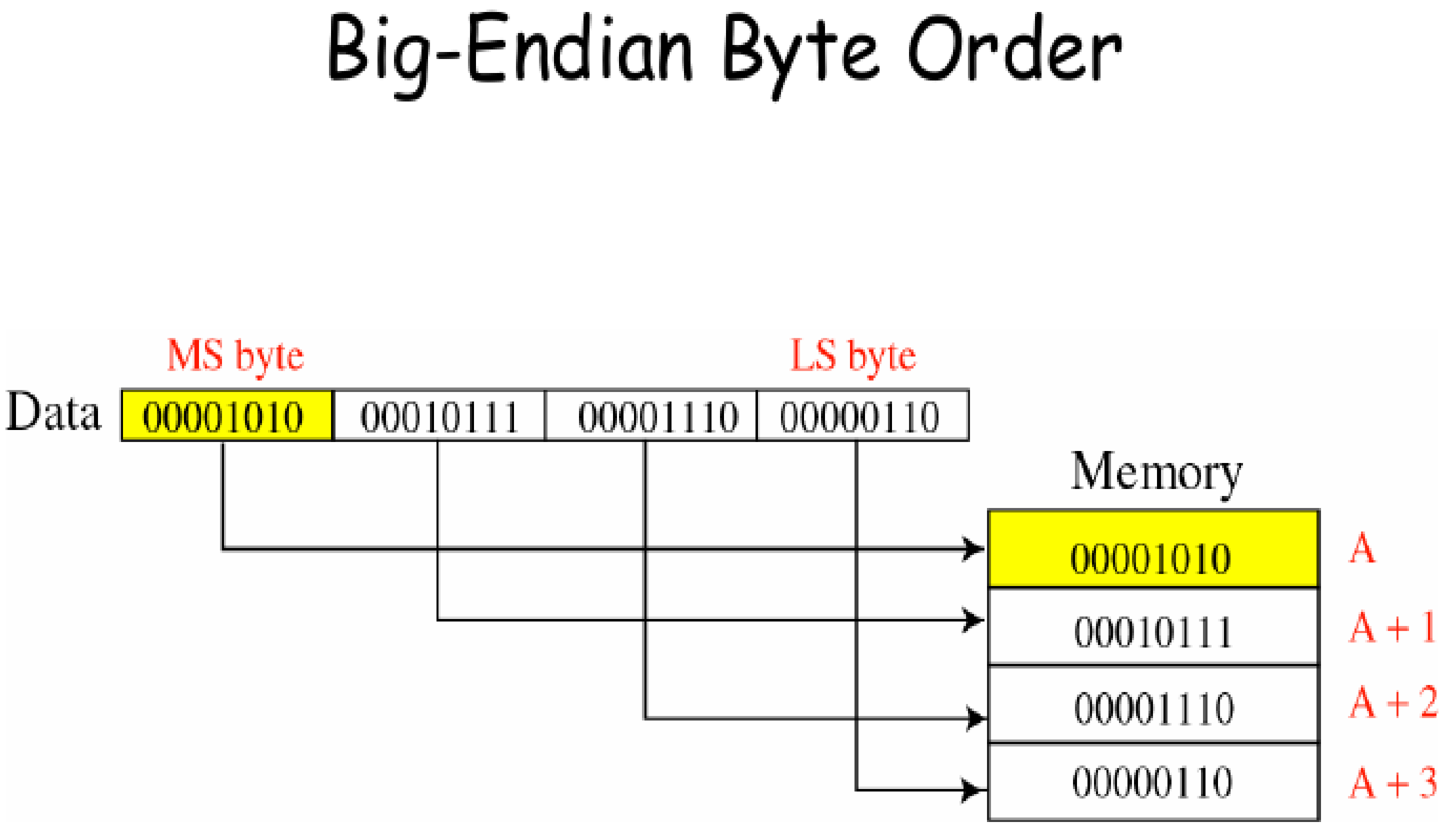
SOL_TCP	*Purpose*
TCP_MAXSEG	Maximum segment size
TCP_NODELAY	Disable the Nagle algorithm

- See "man 7 socket", "man 7 ip", "man 7 tcp", ...

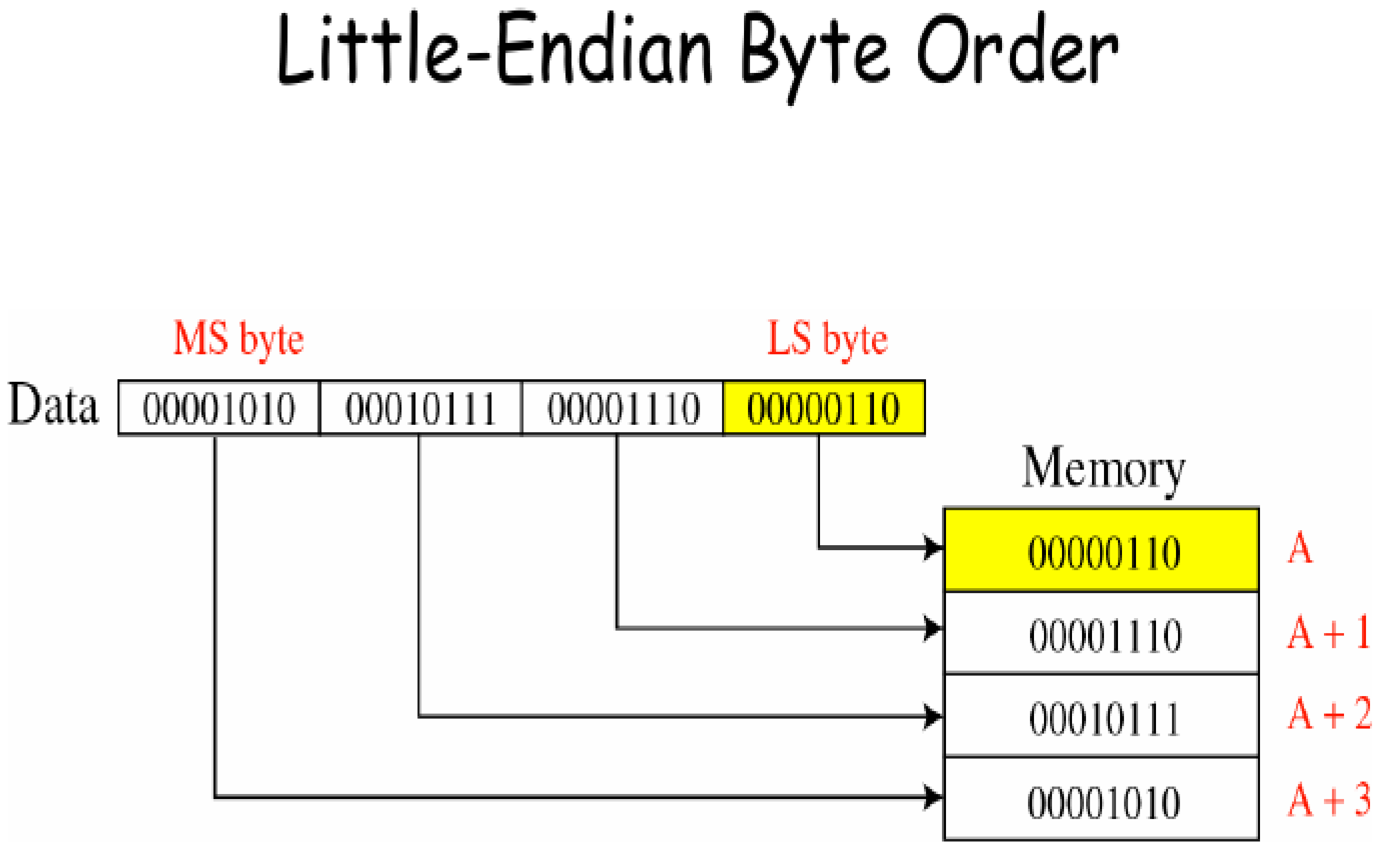
Byte Ordering

- Big Endian vs. Little Endian
  - Little Endian (Intel, DEC):
    - Least significant byte of word is stored in the lowest memory address
  - Big Endian (Sun, SGI, HP):
    - Most significant byte of word is stored in the lowest memory address
  - Network Byte Order = Big Endian
    - Allows both sides to communicate
    - Must be used for some data (i.e. IP Addresses)
    - Good form for all binary data

Big-Endian Format



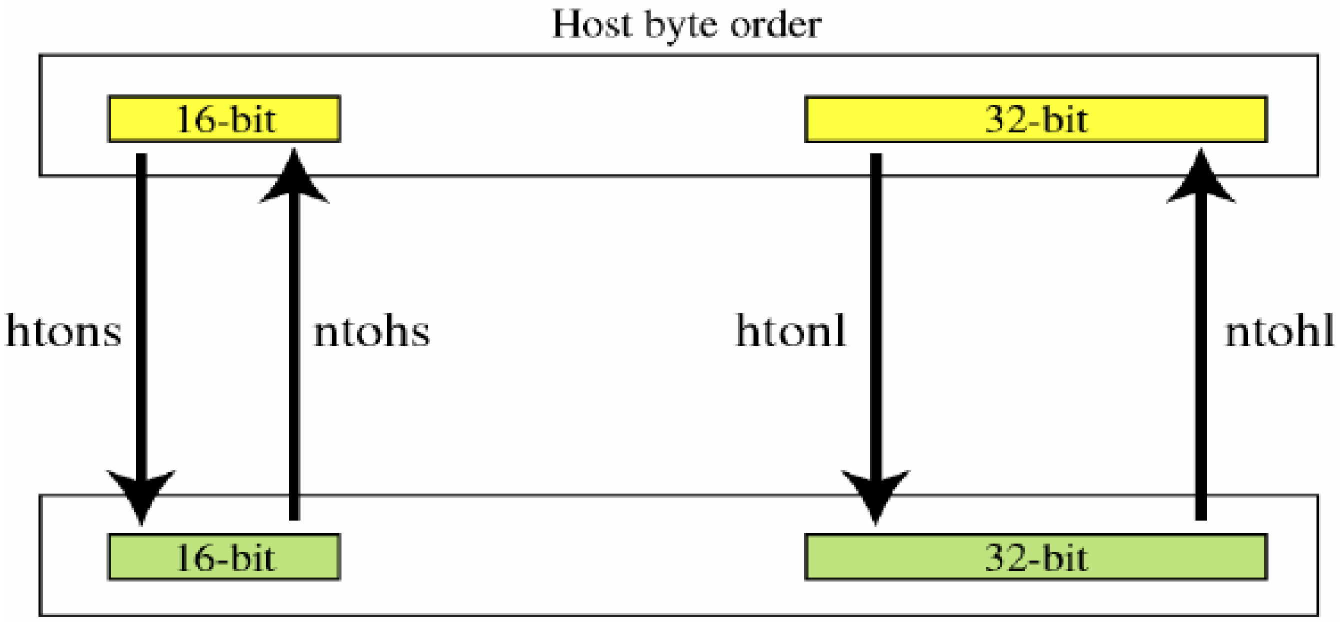
Little Endian Format



Network Transformation

Byte-Order Transformation

- "short"—16 bits
- "long"—32 bits



Byte Ordering Functions

- 16- and 32-bit conversion functions (for platform independence)
- Examples:

```
int m, n;
short int s,t;

m = ntohl (n)   net-to-host long (32-bit) translation
s = ntohs (t)   net-to-host short (16-bit) translation
n = htonl (m)   host-to-net long (32-bit) translation
t = htons (s)   host-to-net short (16-bit) translation
```

Connecting the socket

```
connect(sockfd, (struct sockaddr *) their_addr, sizeof their_addr);
```

- Actually creates a TCP connection, contacting a server
- Afterwards, use file descriptor like a regular file - read, write, close
- Return values (Error):
  - ETIMEDOUT: Client TCP does not receive a response to the SYN packet
  - ECONNREFUSED: No process is waiting on the server at the port specified
  - EHOSTUNREACH: If the client’s SYN packet gets an ICMP “destination unreachable” message

bind

```
bind(sockfd, (struct sockaddr *) my_addr, sizeof my_addr);
```

- Connects a socket with a well-known *incoming* port on a local system
- Common error value:
  - EADDRINUSE – Address / port already in use

Listen

```
listen(sockfd, 10);
```

- Start accepting connections
- 10 is the amount of backlog connections
  - If more than 10 clients waiting, computer will refuse new connections
- How is the backlog arrived at?

Backlog entries????

- Backlog = Entries in incomplete connection queue + completed connection queue

Figure 4.7. The two queues maintained by TCP for a listening socket.

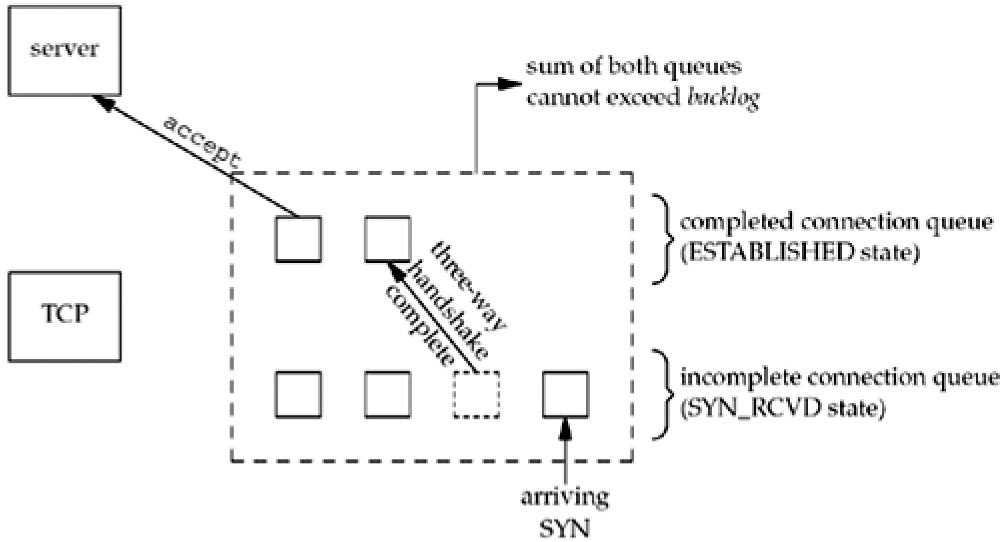
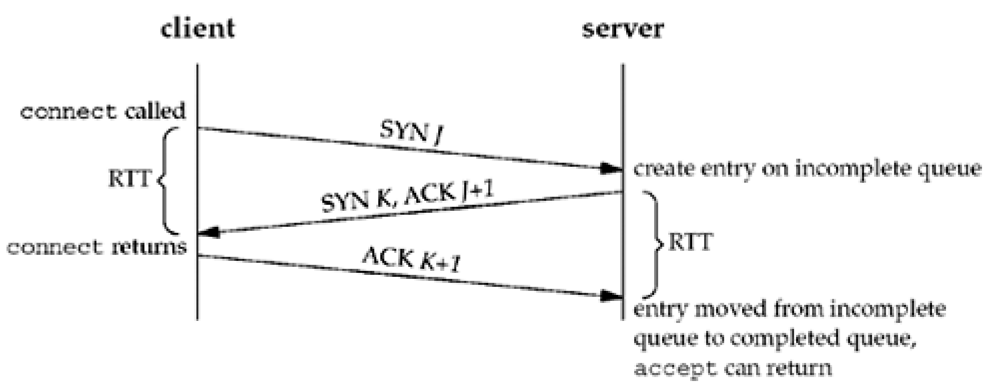


Figure 4.8. TCP three-way handshake and the two queues for a listening socket.



Accept

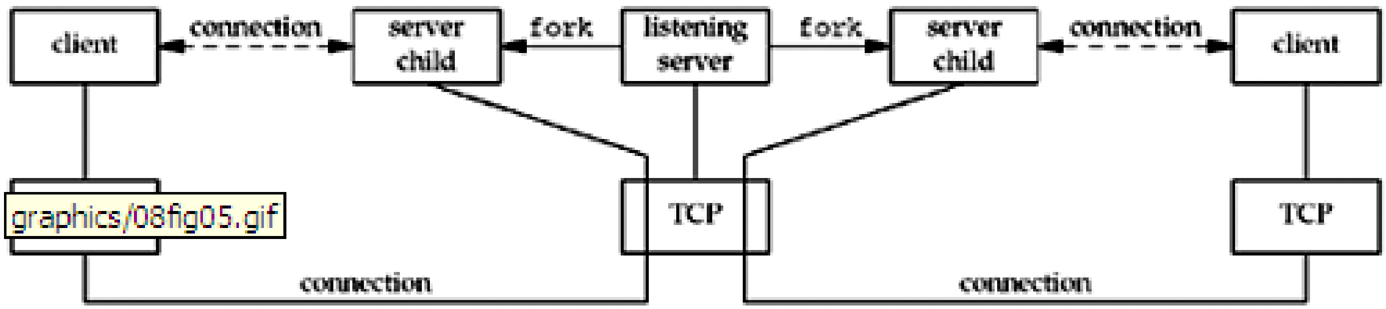
```
int newfd = accept(sockfd, (struct sockaddr *) & their_addr,
                    &sin_size);
```

- Receive a new connection
  - their_addr contains the address of the remote host
- newfd is the socket descriptor for communications
  - Used like a regular socket, can read write close

Outline of a typical concurrent server

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

    /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept (listenfd, ... );    /* probably blocks */

    if( (pid = Fork()) == 0) {
        Close(listenfd);    /* child closes listening socket */
        doit(connfd);       /* process the request */
        Close(connfd);      /* done with this client */
        exit(0);            /* child terminates */
    }

    Close(connfd);          /* parent closes connected socket */
}
```

Summary of TCP client/server with two clients.



Summary of UDP client/server with two clients.

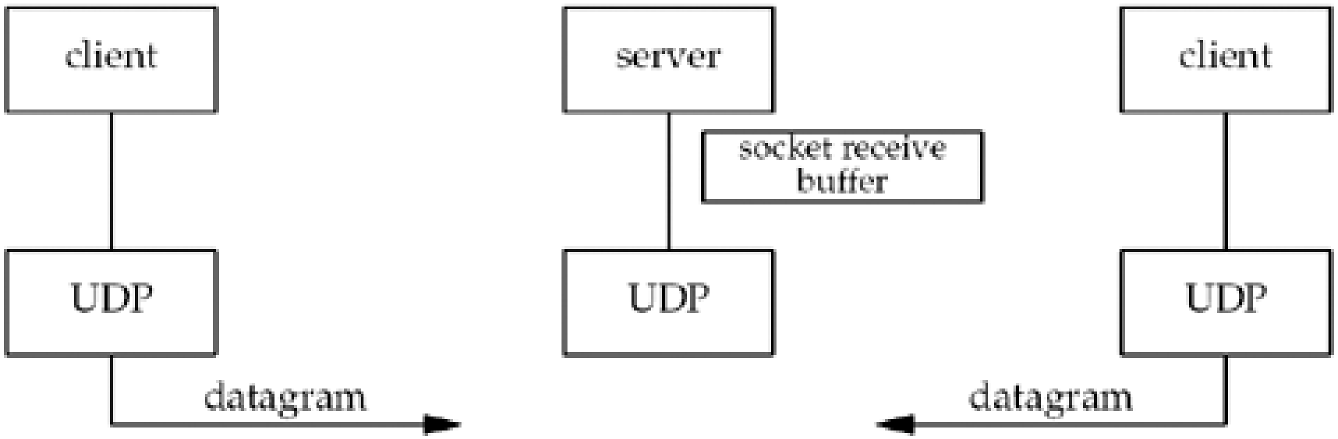


Figure 4.14. Status of client/server before call to `accept` returns.

Figure 4.15. Status of client/server after return from `accept`.

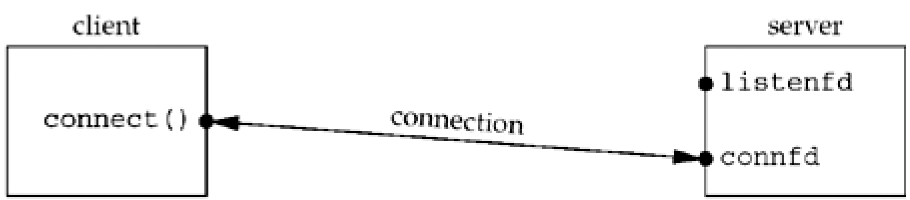


Figure 4.16. Status of client/server after `fork` returns.

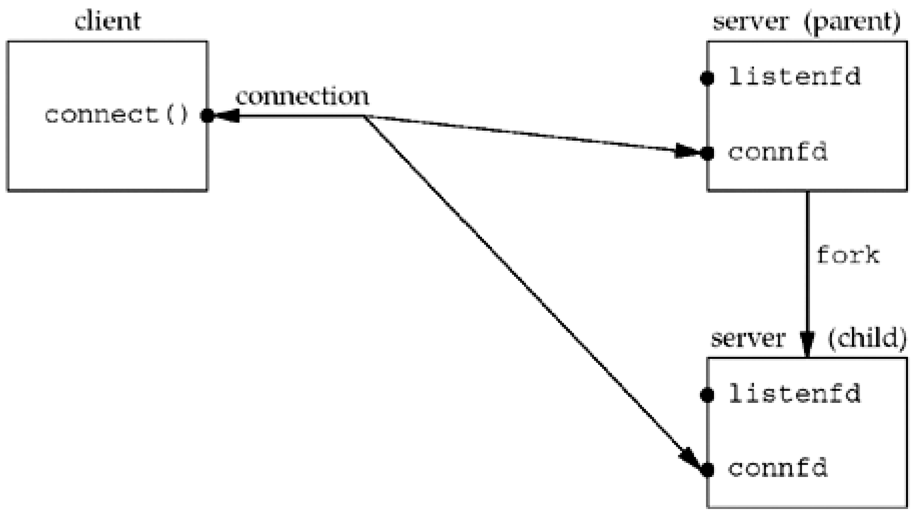
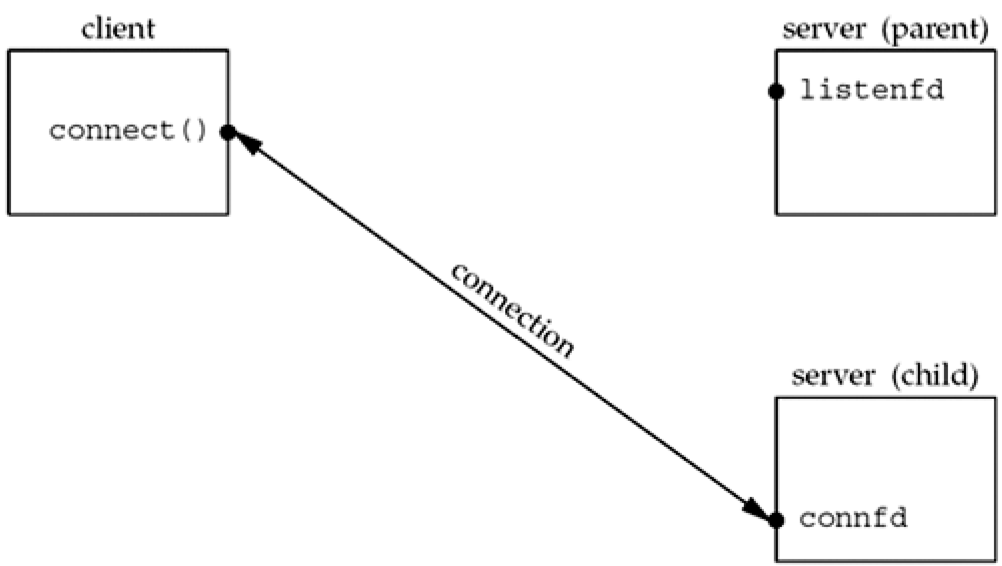


Figure 4.17. Status of client/server after parent and child close appropriate sockets.



Server

```
int main(int argc, char **argv) {
  int sockfd = socket(PF_INET, SOCK_STREAM, 0);
  struct sockaddr_in my_addr;
 my_addr.sin_addr = INADDR_ANY;
 my_addr.sin_port = htons(atoi(argv[1]));
 my_addr.sin_family = AF_INET;
 bind(sockfd, (struct sockaddr *) my_addr, sizeof my_addr);
  listen(sockfd, 10);
  while (1) {
    struct sockaddr_in their_addr;
    int sin_size;
    int newfd = accept(sockfd, (struct sockaddr *) & their_addr,
    &sin_size);
    write(newfd, “Hello world!\n”, 14);
    close(newfd);
} }
```

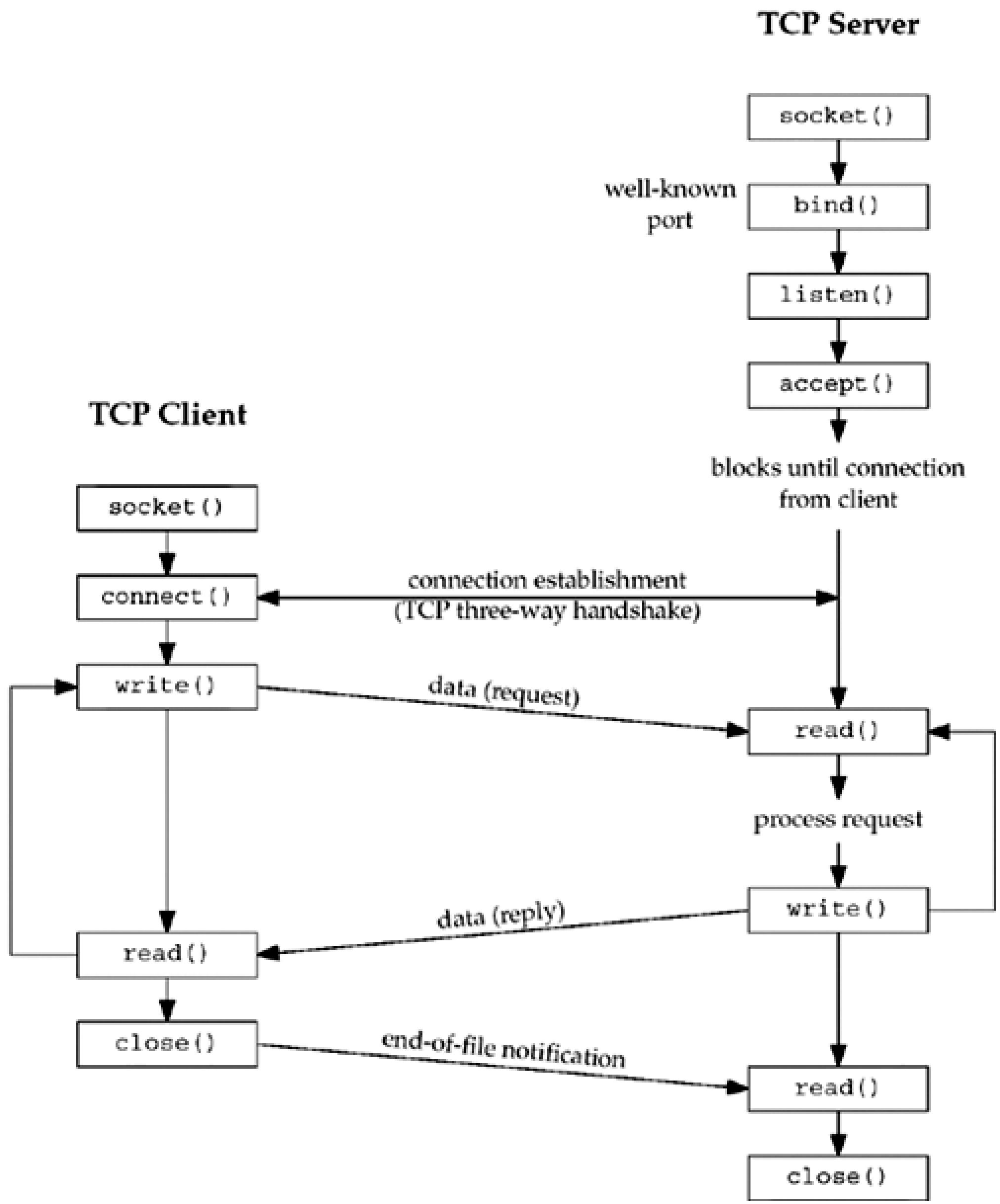

TCP Server
socket()
well-known
port
bind()
listen()
accept()
blocks until connection
from client
TCP Client
socket()
connect()
connection establishment
(TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

Figure 8.1. Socket functions for UDP client/server.

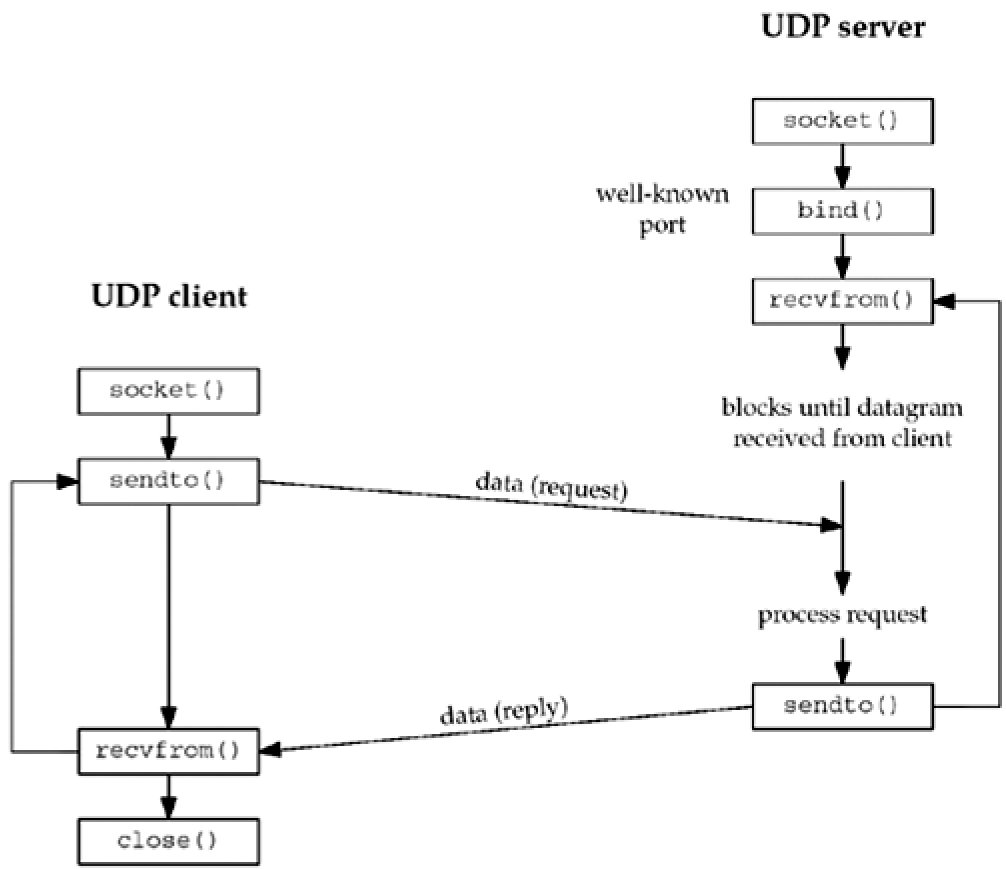


Figure 8.2. Simple echo client/server using UDP.

Support Slides

Programming Interfaces for network programs

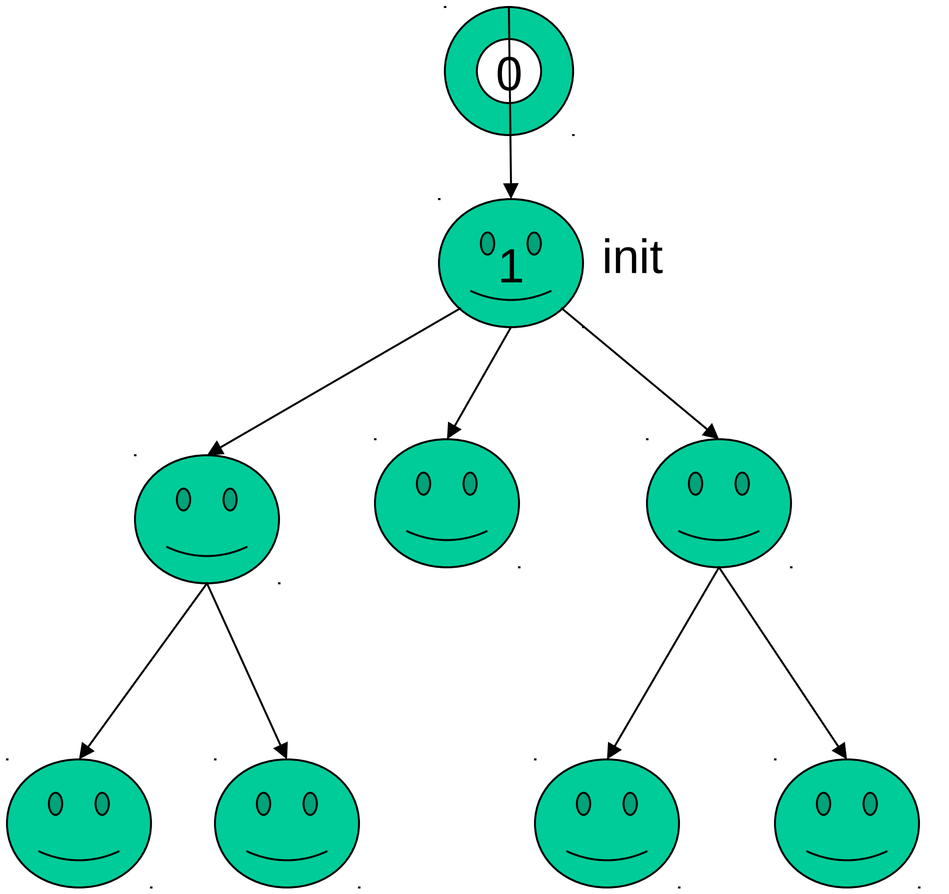
UNIX Process & Socket programming

Process Control

- Outline
- process management
- Fork() and Exec() systems calls
- error handling: perror

Process Trees

- Only an existing process can create a new process
- Parent-Child relations



What is Process?

- An entry in the kernel’s process table

- Most common unit of execution
- Execution state
- Machine instructions, data and environment

Unix Process Model

- Simple and powerful primitives for process creation and initialization.
  - ***fork*** syscall creates a *child* process as (initially) a clone of the parent
  - parent program runs in child process to set it up for ***exec***
  - child can ***exit***, parent can ***wait*** for child to do so.
- Rich facilities for controlling processes by asynchronous *signals*.
  - notification of internal and/or external events to processes or groups
  - the look, feel, and power of interrupts and exceptions
  - default actions: stop process, kill process, dump core, no effect
  - user-level handlers

Process State

- **Process States**
  - ***Running:*** *the process is currently using CPU*
  - ***Runnable:*** *the process can make use of the CPU*
  - ***Sleeping:*** *the process is waiting for an event, e.g. I/O, to occur*
  - ***Suspended:*** *the process is frozen by a signal; another signal is needed to wake up*
  - ***Idle:*** *the process is being created by a fork()and not yet runnable*
  - ***Zombified:*** *the process ended but not returned its exit code to its parent; a process remain zombie until its parent accepts its return code via the wait() system call*

Unix Process Ceation

- Every process, except process 0, is created by the fork() system call
  - fork() allocates entry in process table and assigns a unique PID to the child process
  - child gets a copy of process image of parent: both child and parent are executing the same code following fork()
  - but fork() returns the PID of the child to the parent process and returns 0 to the child process

Unix Process Control

*The **fork** syscall returns a zero to the child and the child process ID to the parent.*

```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        …..
        pid = wait(&status);
} else {
        /* child */
        …..
        exit(status);
}
```

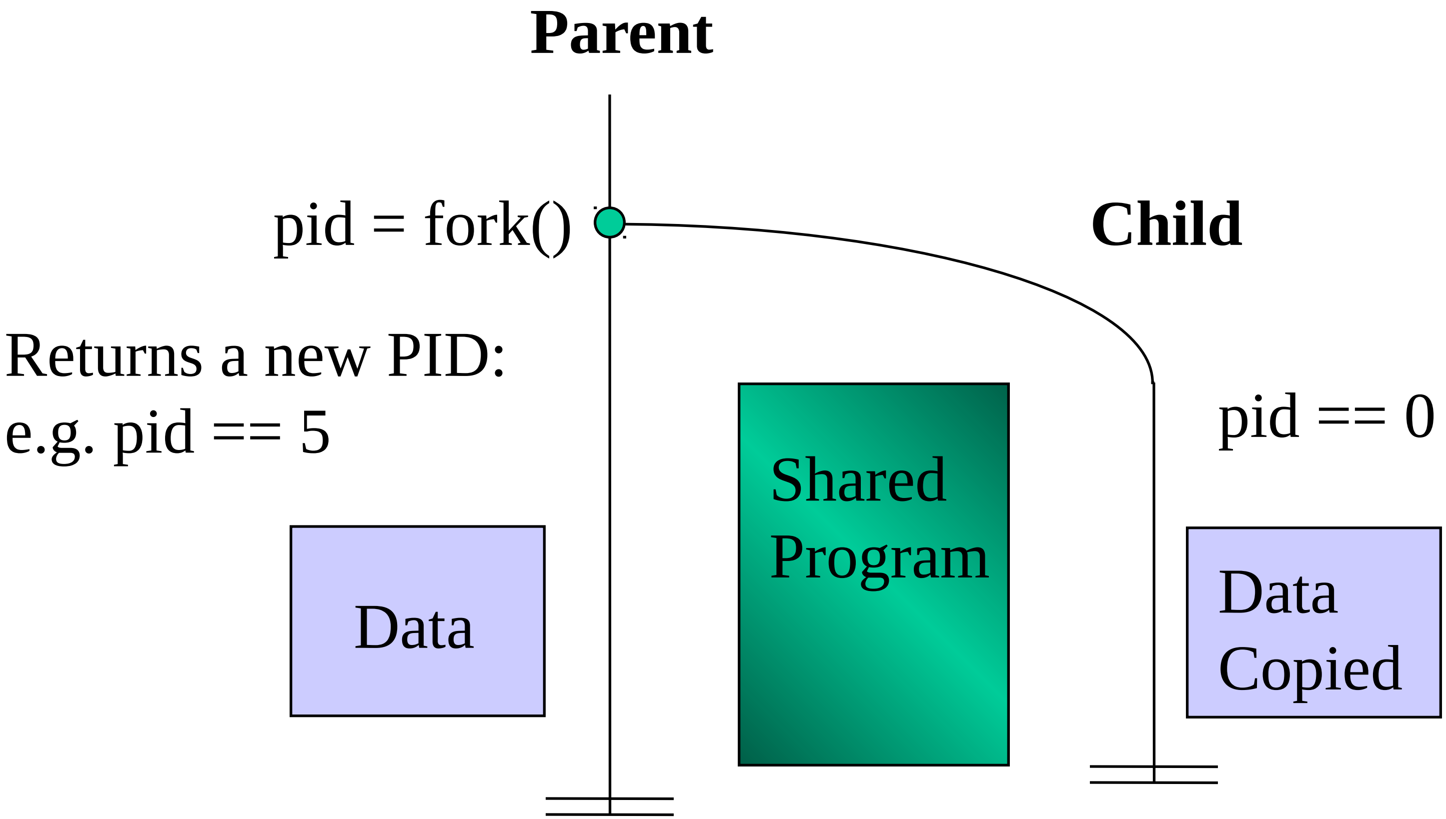
***Fork** creates an exact copy of the parent process.*

*Parent uses **wait** to sleep until the child exits; wait returns child pid and status.*

***Wait** variants allow wait on a specific child, or notification of stops and other signals.*

*Child process passes status back to parent on **exit**, to report success/failure.*

fork() as a diagram



Child Discipline

- After a *fork*, the parent program has complete control over the behavior of its child process.
- The child inherits its execution environment from the parent...but the parent *program* can change it.
  - sets bindings of file descriptors with *open*, *close*, *dup*
  - *pipe* sets up data channels between processes
- Parent program may cause the child to execute a different program, by calling *exec** in the child context.

Fork usage Key Points

- Parent and child both run same code
- Distinguish parent from child by return value from fork
- Start with same state, but each has private copy
- Including shared output file descriptor
- Relative ordering of their print statements undefined

Example using fork()

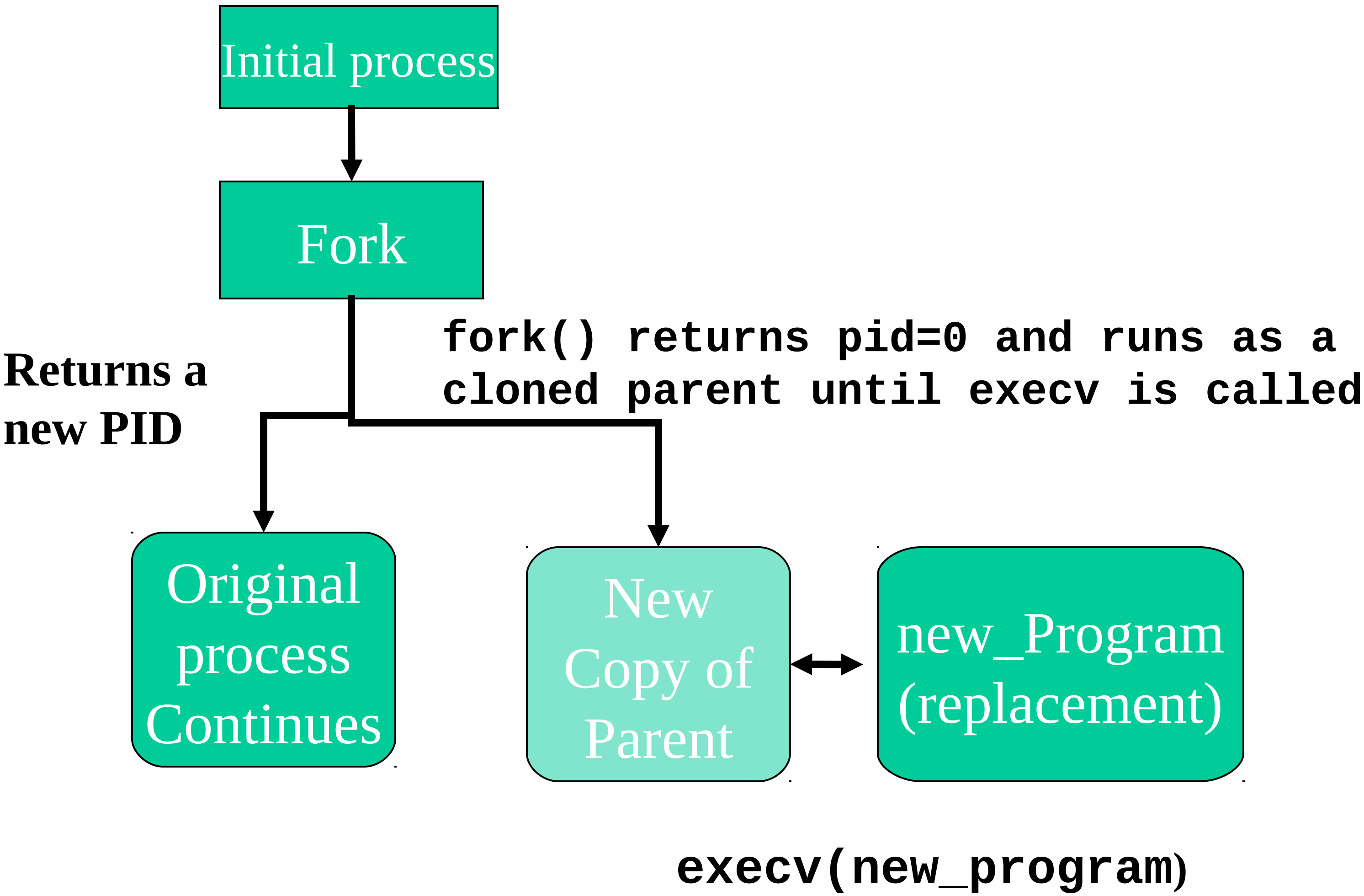
```
void fork1()
{
    intx = 1;
    pid_tpid= fork();
    if (pid== 0)
    {
        printf("Childhas x = %d\n", ++x);
    }
    else
    {
        printf("Parenthas x = %d\n", --x);
    }
    printf("Byefrom process %d with x = %d\n", getpid(), x);
}
```

Exec, Execve, etc.

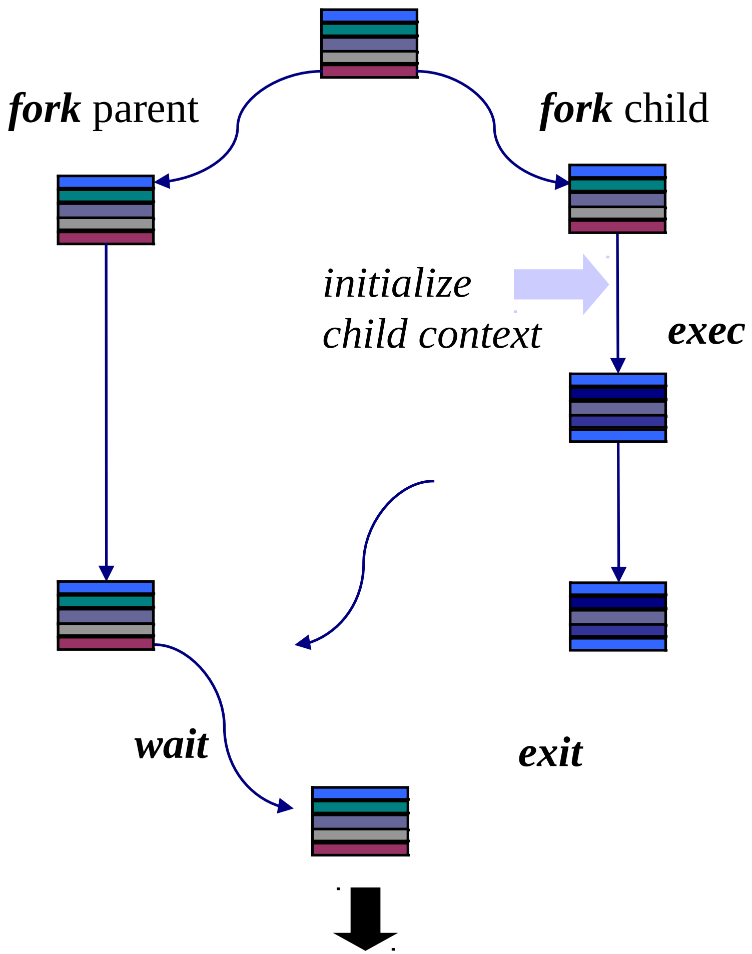
- Children should have lives of their own.
- Exec* “boots” the child with a different executable image.
  - parent program makes *exec** syscall (in forked child context) to run a program in a new child process
  - *exec** overlays child process with a new executable image
  - restarts in user mode at predetermined entry point (e.g., *crt0*)
  - no return to parent program (it’s gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged

fork() and execv()

- **`execv(new_program, argv[ ])`**



Fork/Exec/Exit/Wait Example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process.

int pid = wait*(&status);

Wait for exit (or other status change) of a child.

exec(..) Family

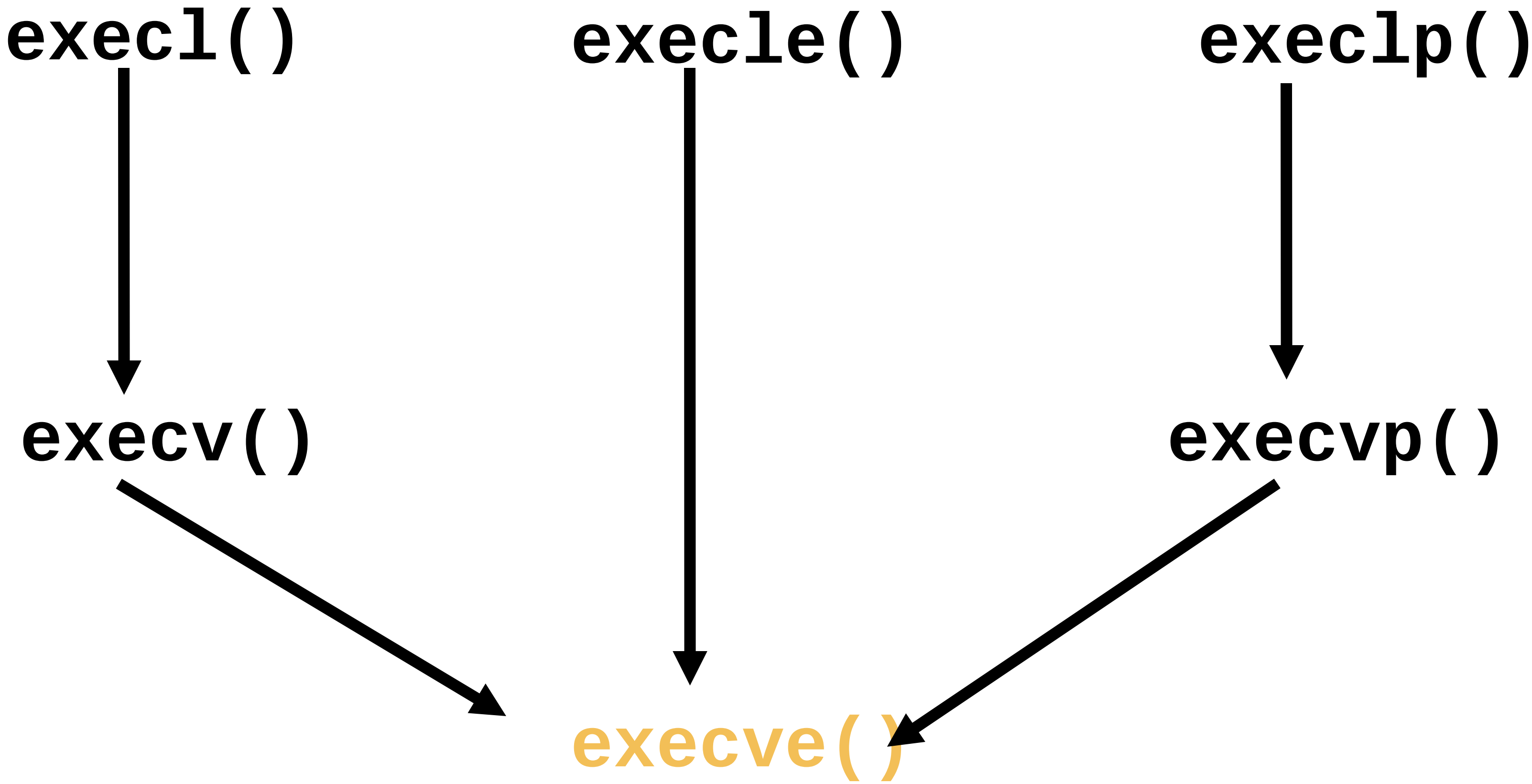
- There are 6 versions of the exec function, and they all do about the same thing: they replace the current program with the text of the new program. Main difference is how parameters are passed.

Versions of Exec

- `int execl( const char *path, const char *arg, ... );`
- `int execlp( const char *file, const char *arg, ... );`
- `int execle( const char *path, const char *arg`
- `          , ..., char *const envp[] );`

- `int execv( const char *path, char *const argv[] );`
- `int execvp( const char *file, char *const argv[] );`
- `int execve( const char *filename, char *const argv[], char *const envp[] );`

exec(..) Family



Execve Example

```
int pid;
int status = 0;

if (pid = fork()) {
    /* parent */
    .....
    pid = wait(&status);
} else {
    /* child */
    execve(const  char  *filename,          char  *const argv [],
      char *const envp[]);
    // exec shouldn't return
    return(ERROR);
}
```

Exit Usage

- void exit(intstatus)
- exits a process
- Normally return with status 0
- atexit() registers functions to be executed upon exit

```
void cleanup(void)
{
    printf("cleaningup\n");
}
void fork6()
{
    atexit(cleanup);
    fork();
       exit(0);
}
```

Error Handling

- Should always check return code of system calls
  - Not only for 5 marks in your lab!!! ;-)
  - There are subtle ways that things can go wrong
  - Use the status info kernel provides us
- Approach in this class: Wrappers
- Different error handling styles
  - Unix-Style
  - Posix-Style

Perror usage

- Prints system Error Message
- Describes last error encountered during a call to system or library function.

```
char *bufptr;
size_t szbuf;
if ((bufptr = malloc(szbuf)) == NULL)
{
 perror("malloc");
 exit(2);
}
```