Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 3:
Advance Unix Network Programming
(12 hrs)

Outline

1. I/O Models in Unix
   i. Blocking I/O
   ii. Non-blocking I/O
   iii. I/O multiplexing (select(), pselect() poll())
   iv. Signal-driven I/O
   v. Asynchronous I/O

2. Concurrent Server Design
   i. Overview of process and threads
   ii. Fork() and exec() function
   iii. Using fork() to handle multiple clients
   iv. Using select() to handle multiple socket descriptors
   v. Multithreading model using pthreads
3. Implementing broadcast and multicast communication
4. Socket Options
   i. Using setsockopt(), getsockopt(), fcntl() and ioctl() to modify socket behavior
   ii. Common options: SO_REUSEADDR, SO_BROADCAST, SO_KEEPALIVE, SO_LINGER etc.
5. Logging in Unix
   i. Introduction to Syslog
   ii. Logging messages from network applications
   iii. Configuring and using syslog(), openlog(), and closelog()
6. Socket operations
7. Introduction to P2P programming
8. P2P Socket fundamentals
9. Overview Network Security Programming
   i. Defining Security
   ii. Challenges of Security
   iii. Securing by Hostname or Domain Name
   iv. Identification by IP Number
   v. Wrapper program to implement simple security policy

IO Models in Unix

IO Models in Unix

- TCP echo client is handling two inputs at the same time: standard input and a TCP socket
  - when the client was blocked in a call to read, the server process was killed
  - server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
    - We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
    - I/O multiplexing (**select**, **poll**, or newer **pselect** functions)
- Scenarios for I/O Multiplexing
  - client is handling multiple descriptors (interactive input and a network socket).
  - Client to handle multiple sockets (rare)
  - TCP server handles both a listening socket and its connected socket.
  - Server handle both TCP and UDP.
  - Server handles multiple services and multiple protocols

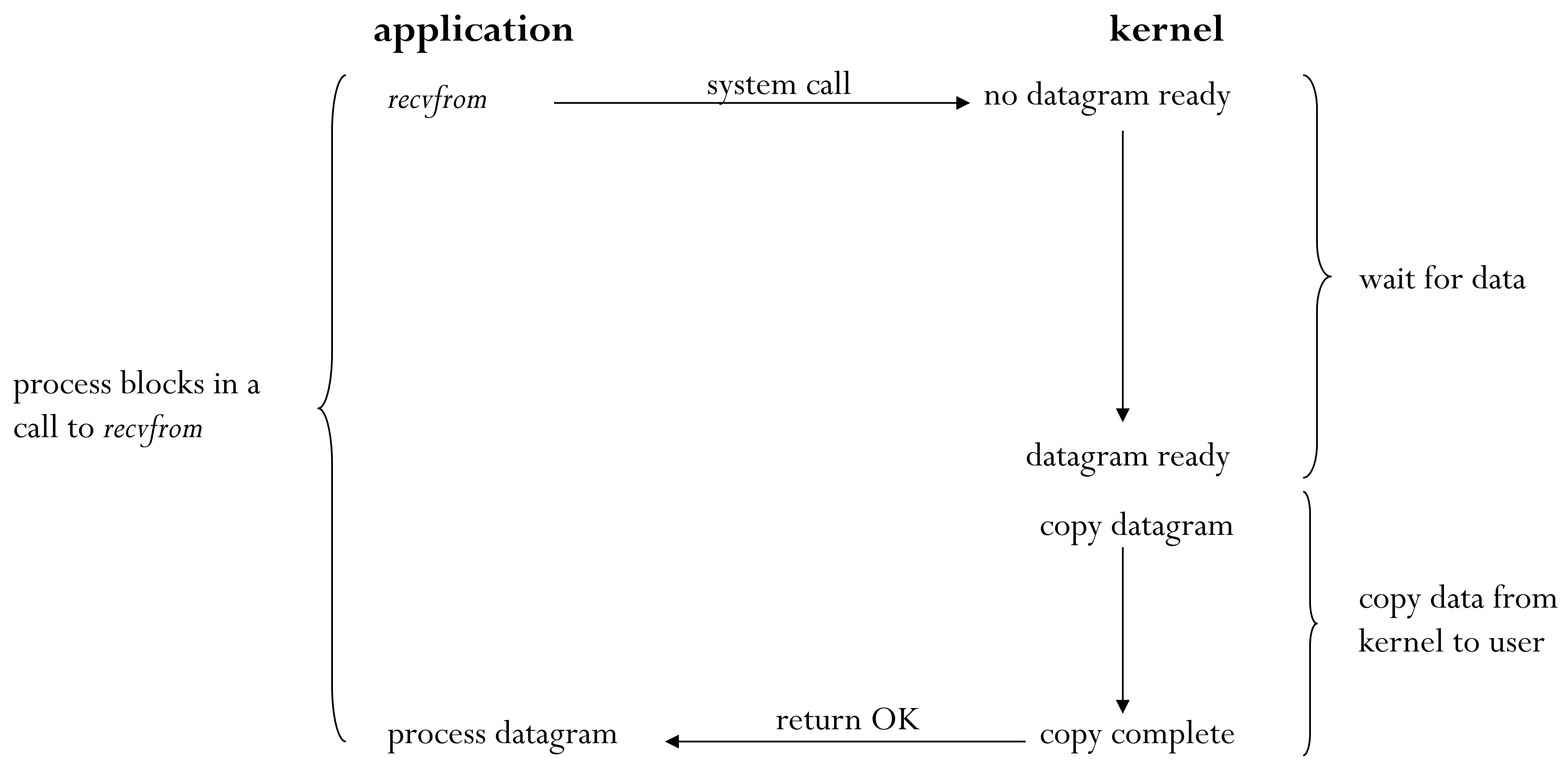
IO Models in Unix

Models

1. Blocking I/O
2. Nonblocking I/O
3. I/O multiplexing(**select** and **poll**)
4. Signal driven I/O (**SIGIO**)
5. Asynchronous I/O

- Two *distinct phases* for an input operation
  - Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
  - Copying the data from the kernel to the process (from kernel buffer into application buffer)

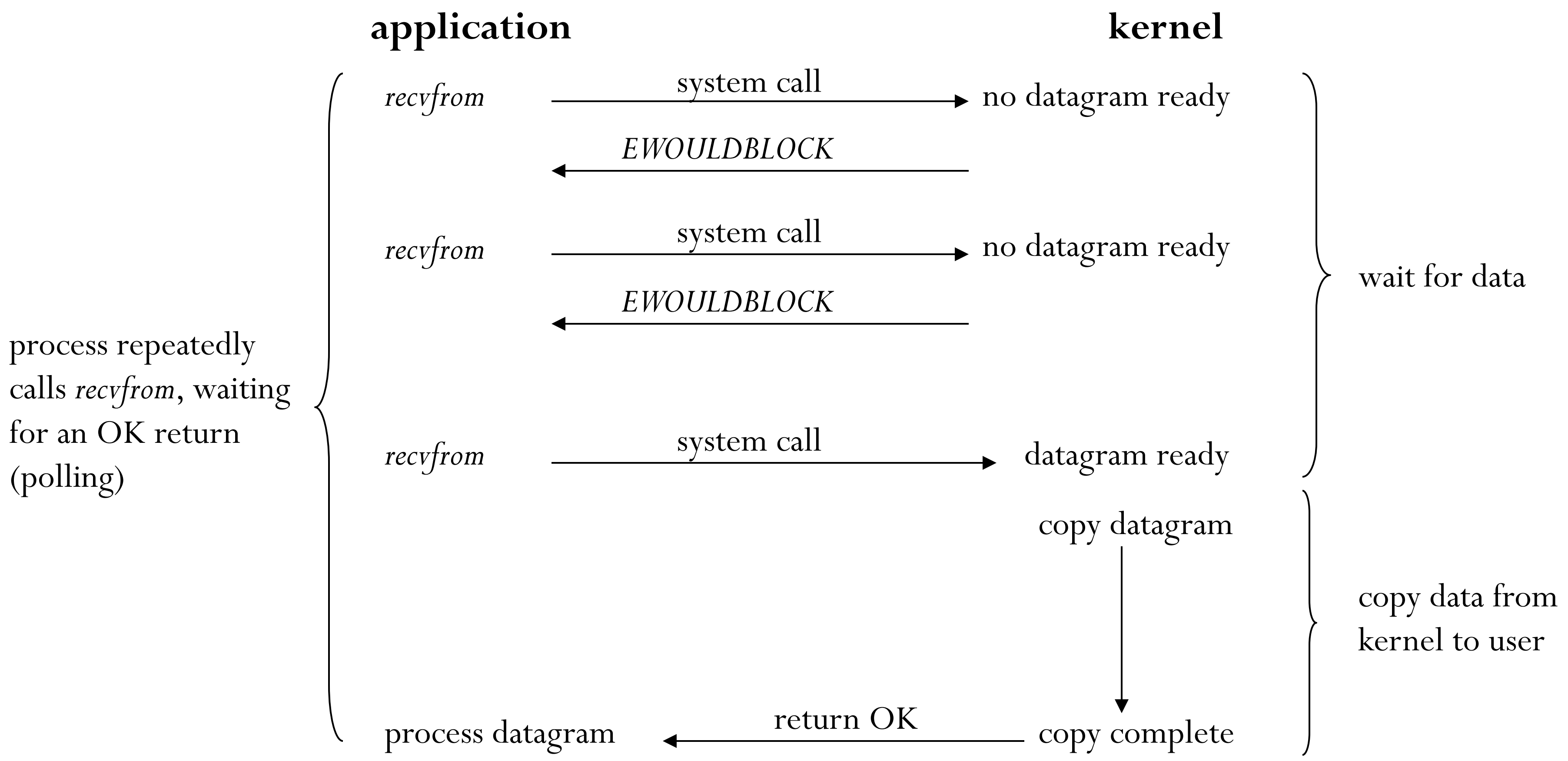
Blocking I/O Model



Blocking I/O model…

- By default, all sockets are blocking.
- The process calls **recvfrom** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- We say that our process is blocked the entre time from when it calls **recvfrom** until it returns.
- When **recvfrom** returns successfully, our application process the datagram.

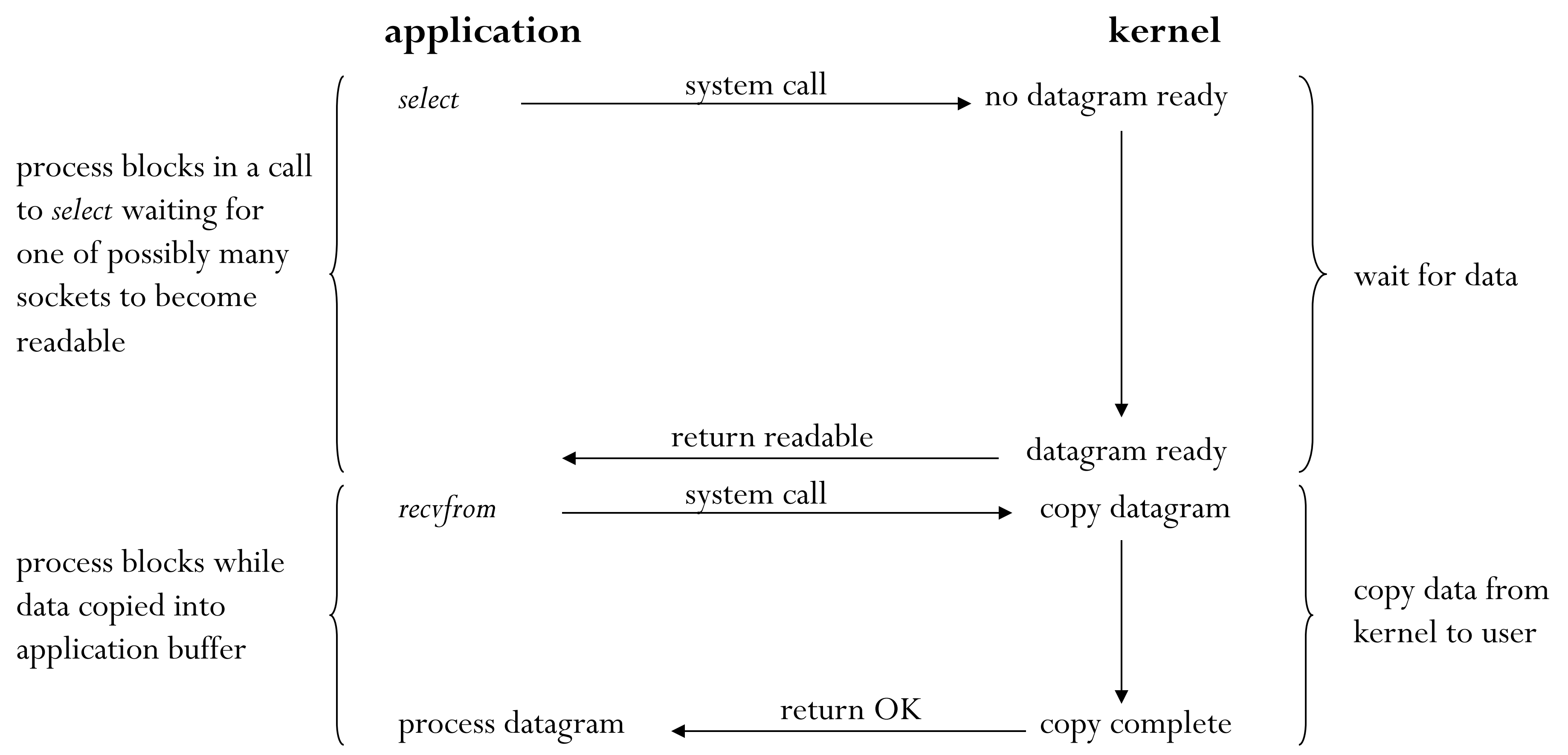
Non-blocking I/O Model



Non-blocking I/O Model…

- When a socket is non-blocking, It instruct the kernel as "when an I/O operation that the process requests cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead."
- The first two times(as in fig) that we call **recvfrom**, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead.
- The third time we call **recvfrom**, a datagram is ready, it is copied into our application buffer, and **recvfrom** returns successfully.
- We then process data. When an application sits in a loop calling **recvfrom** on a non-blocking descriptor like this, it is called polling.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time.

I/O Multiplexing Model


application
kernel
select
system call
no datagram ready
process blocks in a call to *select* waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
process datagram
return OK
copy complete

I/O Multiplexing Model…

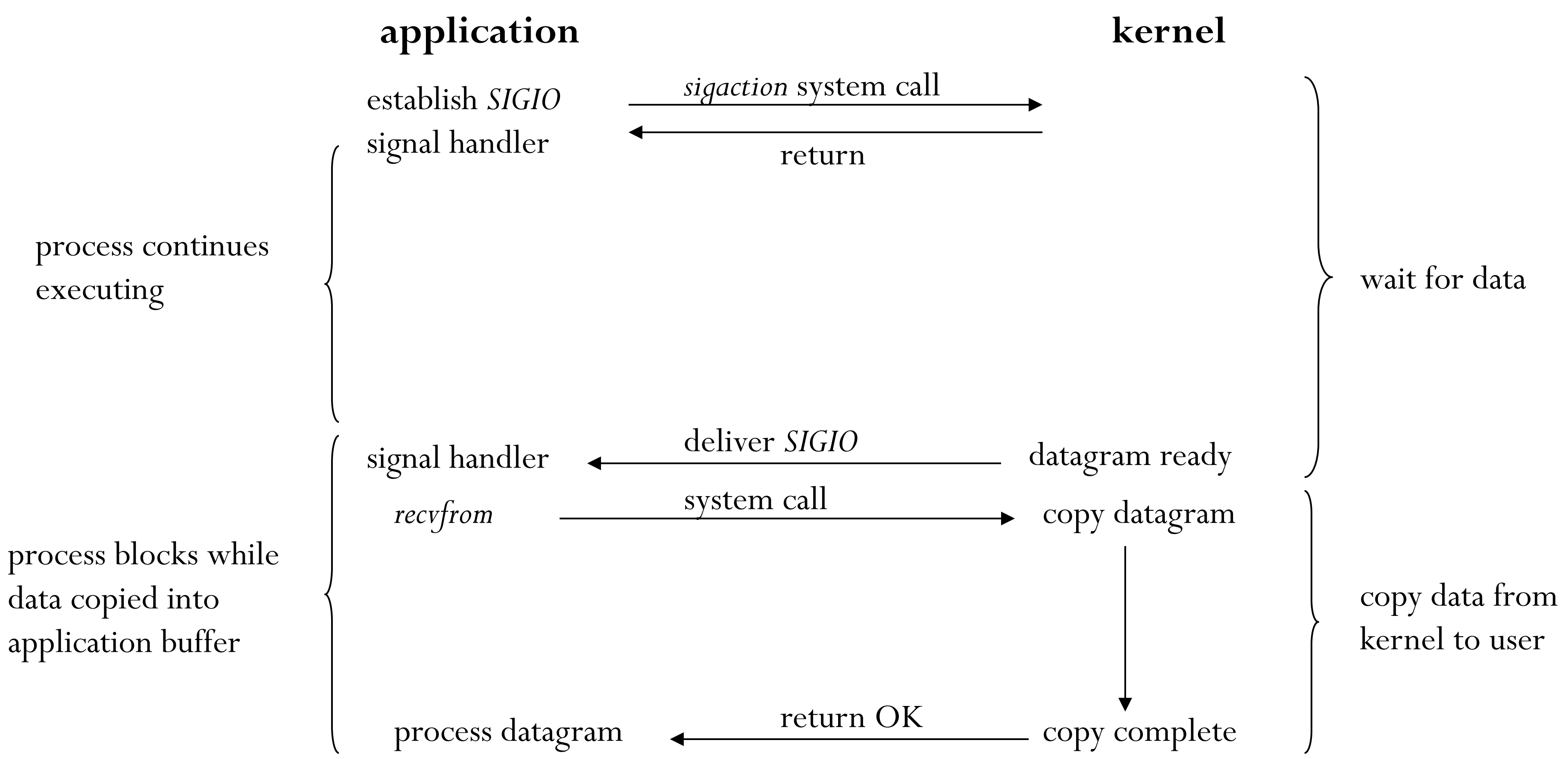
- With I/O multiplexing, we call **select** or **poll** and block in one of these two system calls, instead of blocking in the actual I/O system call.

- We block in a call to **select**, waiting for the datagram socket to be readable.

- When **select** returns that the socket is readable, we then call **recvfrom** to copy the datagram into our application buffer.

- With **select**, we can wait for more than one descriptor to be ready.

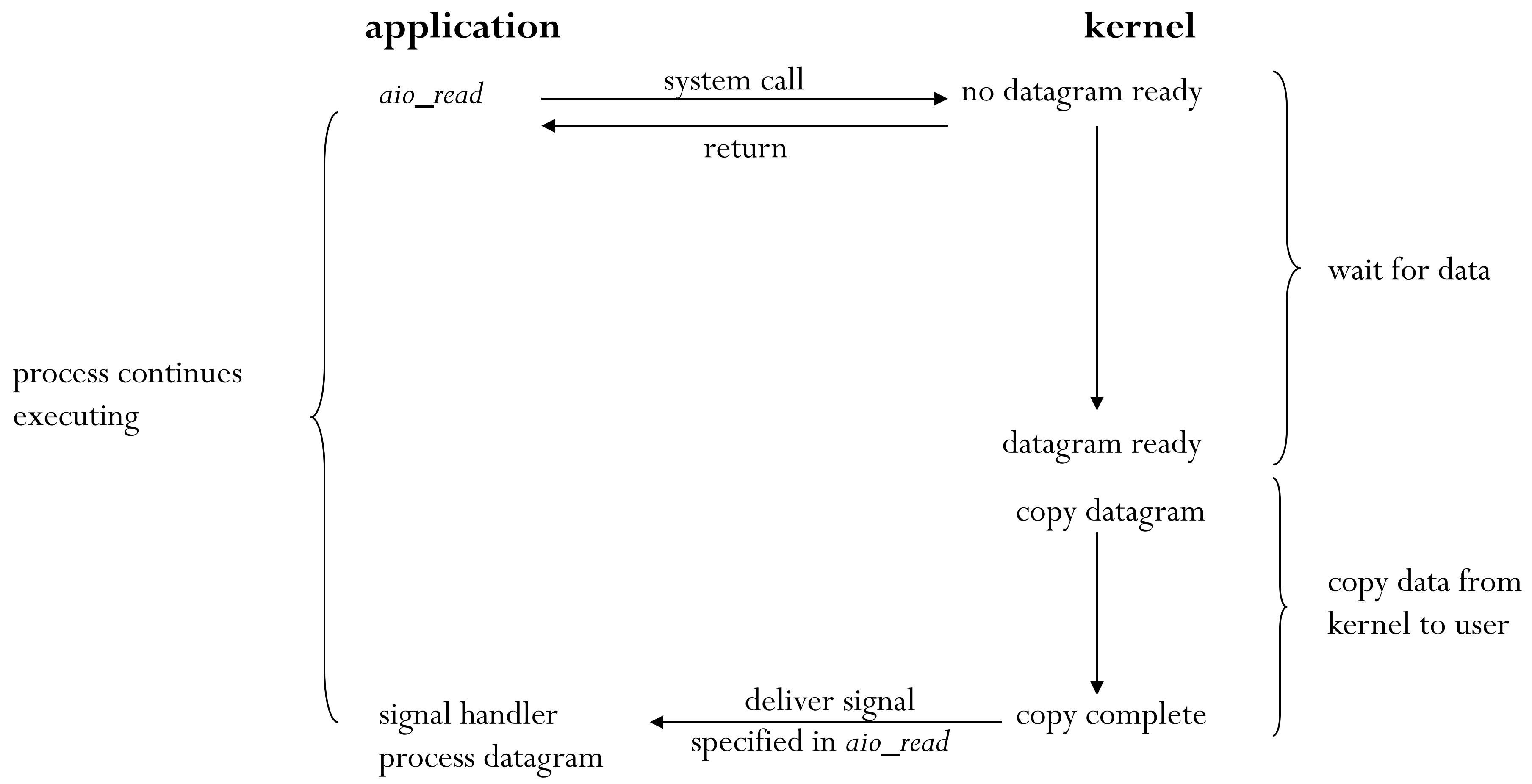
Signal Driven I/O Model



Signal Driven I/O Model…

- A **SIGIO signal** is used to tell the kernel when the descriptor is ready. We call this signal-driven I/O.
- We first enable the socket for the signal-driven I/O and install a signal handler using the **sigaction** system call.
- The return from this system call is immediate and our process continues; it is not blocked.
- When the datagram is ready to be read, the **SIGIO** signal is generated for our process. We can then read the data.
- The advantage of this model is that we are not blocked while waiting for the datagram to arrive.
- The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

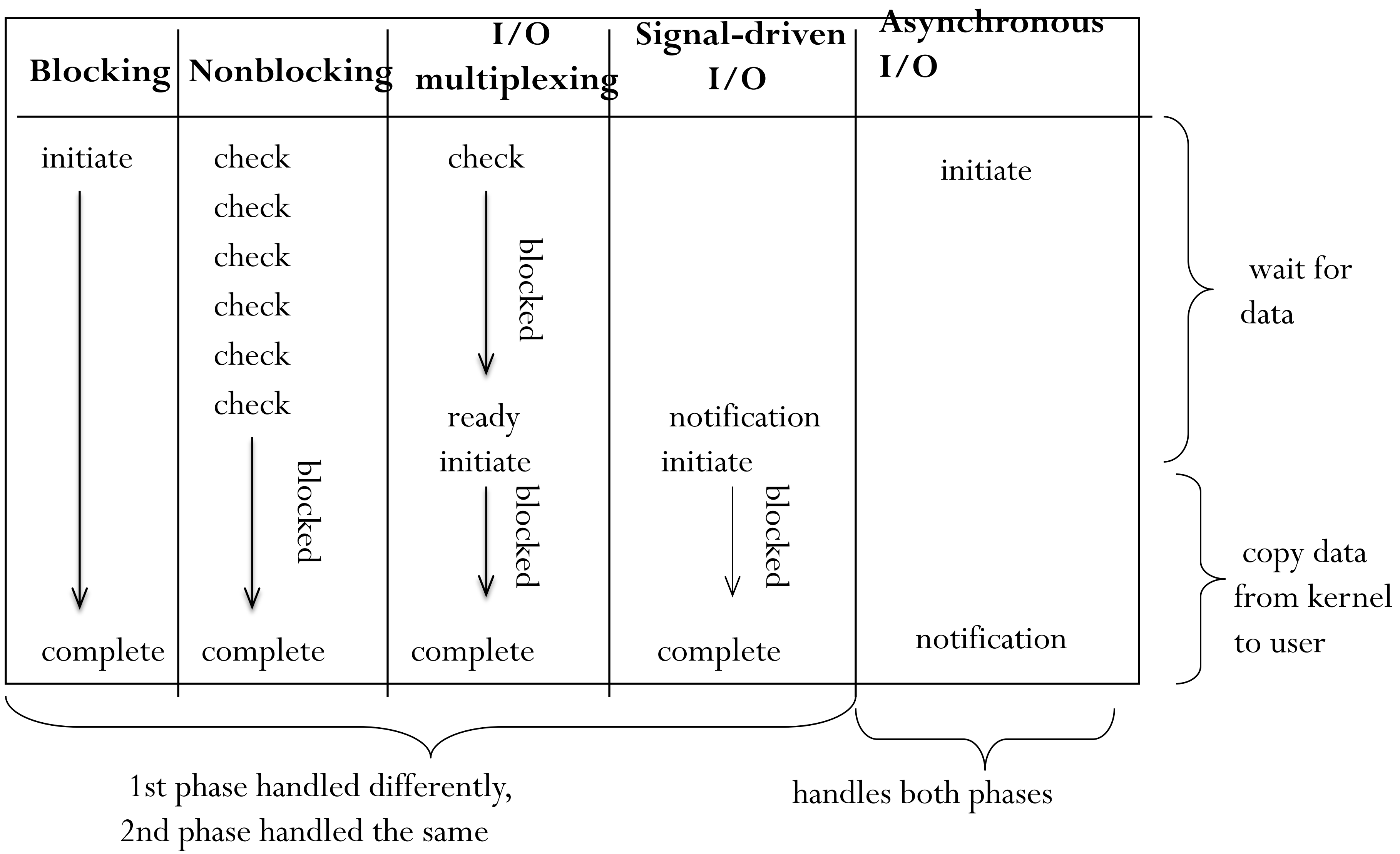
Asynchronous I/O Model


application
kernel
aio_read
system call
no datagram ready
return
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

Asynchronous I/O Model

- Asynchronous I/O is defined by the POSIX specification.
- These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
- The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/ operation is complete.
- We call **aio_read** and pass the kernel the descriptor, buffer pointer, buffer size, file offset, and how to notify us when the entire operation is complete.
- This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

Comparison of the I/O Models



Synchronous I/O and Asynchronous I/O

- Synchronous I/O
  - causes the requesting process to be blocked until that I/O operation (recvfrom) completes. (blocking, nonblocking, I/O multiplexing, signal-driven I/O)
- Asynchronous I/O
  - does not cause the requesting process to be blocked

select function

- Allows the process to instruct the kernel to *wait for any one of multiple events to occur* and to wake up the process only when one or more of these events occurs or *when a specified amount of time has passed*.
- What descriptors we are interested in (readable ,writable , or exception condition) and how long to wait?

```
#include <sys/select.h>
 #include <sys/time.h>
 int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
  *exceptset, const struct timeval *);
//Returns: +ve count of ready descriptors, 0 on timeout, -1 on error
struct timeval{
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */ }
```

- The final argument, timeout, tells the kernel how long to wait for one of the specified file descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

Possibilities for select function

1. Wait forever : return only when descriptor (s) is ready (specify **timeout** argument as NULL)
2. wait up to a fixed amount of time: Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. Do not wait at all : return immediately after checking the descriptors(called Polling) (specify **timeout** argument as pointing to a **timeval** structure where the timer value is 0)

- The wait is normally interrupted if the process catches a signal and returns from the signal handler
  - **select** might return an error of **EINTR**
  - Actual return value from function = -1

Return value of select

- Select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred.
- If the time limit expires, select() returns 0.
- If select() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and the global variable **errno** will be set to indicate the error.

errno Constant	Description
EBADF	One or more of the file descriptors is not valid.
EINTR	A signal was caught during the select() call.
EINVAL	One of the arguments is not valid (e.g., nfds is negative, or the timeout struct is invalid).
ENOMEM	Unable to allocate memory for internal tables. Rare.

select function Descriptor Arguments

- **readset →** descriptors for checking readable
- **writeset →** descriptors for checking writable
- **exceptset →** descriptors for checking exception conditions (2 exception conditions)
  - ✓ arrival of out of band data for a socket
  - ✓ the presence of control status information to be read from the master side of a pseudo terminal (Ignore)
- If you pass the 3 arguments as NULL, you have a high precision timer than the sleep function

Descriptor Sets

- Array of integers : each bit in each integer correspond to a descriptor (**fd_set**)
- 4 macros
  - `void FD_ZERO(fd_set *fdset); /* clear all bits in fdset */`
  - `void FD_SET(int fd, fd_set *fdset); /*turn on the bit for fd in fdset */`
  - `Void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset*/`
  - `int FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */`

Example of Descriptor sets Macros

```
fd_set rset;

FD_ZERO(&rset);   /*all bits off : initiate*/
FD_SET(1, &rset);     /*turn on bit fd 1*/
FD_SET(4, &rset);     /*turn on bit fd 4*/
FD_SET(5, &rset);     /*turn on bit fd 5*/
```

maxfdp1 argument to select function

- ❖ specifies the number of descriptors to be tested.
- ❖ Its value is the maximum descriptor to be tested, plus one. (hence maxfdp1)
  - ➢ Descriptors 0, 1, 2, up through and including **maxfdp1**-1 are tested
  - ➢ example: interested in **fds** 1,2, and 5 → **maxfdp1** = 6
  - ➢ Your code has to calculate the **maxfdp1** value constant **FD_SETSIZE** defined by including **<sys/select.h>**
  - ➢ is the number of descriptors in the **fd_set** datatype. (often = 1024)

Value-Result arguments in select function

❖Select modifies descriptor sets pointed to by **readset**, **writeset**, and **exceptset** pointers

- ➢ On function call : Specify value of descriptors that we are interested in
- ➢ On function return : Result indicates which descriptors are ready
- ➢ Use **FD_ISSET** macro on return to test a specific descriptor in an **fd_set** structure
  - ➢Any descriptor not ready will have its bit cleared
  - ➢You need to turn on all the bits in which you are interested on all the descriptor sets each time you call **select**

Condition for a socket to be ready for select

Condition	Readable?	writable?	Exception?
Data to read	●		
read-half of the connection closed	●		
new connection ready for listening socket	●		
Space available for writing		●	
write-half of the connection closed		●	
Pending error	●	●	
TCP out-of-band data			●

pselect() function

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timespec *timeout, const sigset_t
*sigmask);
/* Returns: count of ready descriptors, 0 on timeout, -1 on error */
```

- pselect contains two changes from the normal select function:
- **pselect** uses the **timespec** structure (another POSIX invention) instead of the **timeval** structure. The **tv_nsec** member of the newer structure specifies nanoseconds, whereas the **tv_usec** member of the older structure specifies microseconds.

```
struct timespec {
        time_t tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds */
};
```

- **pselect** adds a sixth argument: a pointer to a signal mask.
  - This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

```
if (intr_flag)
        handle_intr(); /* handle the signal */
/* signals occurring in here are lost */
if ( (nready = select( ... )) < 0) {
        if (errno == EINTR) {
                if (intr_flag)
                handle_intr();
        }
... }
```

- The problem is that between the test of **intr_flag** and the call to select, if the signal occurs, it will be lost if select blocks forever.

```
sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
 /* block SIGINT */
if (intr_flag)
        handle_intr(); /* handle the signal */
 if ( (nready = pselect ( ... , &zeromask)) < 0) {
        if (errno == EINTR) {
                if (intr_flag) handle_intr ();
        }
... }
```

Before testing the **intr_flag** variable, we block SIGINT. When **pselect** is called, it replaces the signal mask of the process with an empty set (i.e., zeromask) and then checks the descriptors, possibly going to sleep. But when pselect returns, the signal mask of the process is reset to its value before pselect was called (i.e., SIGINT is blocked).

poll() Function

```
#include <poll.h>
 int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- returns count of ready descriptors, 0 on timeout, -1 on error.
- poll() examines a set of file descriptors to see if some of them are ready for I/O or if certain events have occurred on them.
- The ***fds*** argument is a pointer to an array of ***pollfd*** structures.
- The ***nfds*** argument specifies the size of the ***fds*** array.
- Each element is a ***pollfd*** structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd {
        int fd; /* descriptor to check */
        short events; /* events of interest on fd */
        short revents; /* events that occurred on fd */
};
```

- **fd**: File descriptor to poll.
- **events**: Events to poll for.
- **revents**: Events which may occur or have occurred.
- The event bitmasks in **events** and **revents** have the following bits:
  - **POLLERR** : An exceptional condition has occurred on the device or socket. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLHUP**: The device or socket has been disconnected. This flag is output only, and ignored if present in the input events bitmask. Note that POLLHUP and POLLOUT are mutually exclusive and should never be present in the revents bitmask at the same time.
  - **POLLIN** : Data other than high priority data may be read without blocking.
  - **POLLNVAL** : The file descriptor is not open. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLOUT** : Normal data may be written without blocking. This is equivalent to POLLWRNORM.
  - **POLLPRI** : High priority data may be read without blocking.
  - **POLLRDBAND** : Priority data may be read without blocking.
  - **POLLRDNORM** : Normal data may be read without blocking.
  - **POLLWRBAND** : Priority data may be written without blocking.
  - **POLLWRNORM** : Normal data may be written without blocking.

Poll() function…

- With regard to TCP and UDP sockets, the following conditions cause poll to return the specified **revent.**
  - All regular TCP data and all UDP data is considered normal.
  - TCP's out-of-band data is considered priority band.
  - When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
  - The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
  - The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
  - The completion of a nonblocking connect is considered to make a socket writable.

- If timeout is greater than zero, it specifies a maximum interval (in milliseconds) to wait for any file descriptor to become ready.
- If timeout is zero, then poll() will return without blocking.
- If the value of timeout is -1, the poll blocks indefinitely.
- RETURN VALUES
  - poll() returns the number of descriptors that are ready for I/O, or -1 if an error occurred.
  - If the time limit expires, poll() returns 0.
  - If poll() returns with an error, including one due to an interrupted call, the fds array will be unmodified and the global variable errno will be set to indicate the error.
  - **poll() will fail if:**
- [**EAGAIN**] :Allocation of internal data structures fails. A subsequent request may succeed.
- [**EFAULT**] : fds points outside the process's allocated address space.
- [**EINTR**] : A signal is delivered before the time limit expires and before any of the selected events occurs.
- [**EINVAL**]:The nfds argument is greater than OPEN_MAX or the timeout argument is less than -1.

Comparison of select(), pselect() and poll()

Feature	select()	pselect()	poll()
Header File	<sys/select.h>	<sys/select.h>	<poll.h>
Descriptor Limit	Limited by FD_SETSIZE (usually 1024)	Same as select()	No hard limit — supports arbitrary fd numbers
Input Parameter Type	Bitmask sets (fd_set)	Bitmask sets (fd_set)	Array of struct pollfd
Output Mechanism	Modifies fd_set in place	Modifies fd_set in place	Sets revents field in each pollfd
Signal Safety	Interrupted by signals (EINTR)	Atomically blocks/unblocks signals	Interrupted by signals (EINTR)
Signal Masking Option	No	Yes — takes sigset_t *sigmask	No
Precision of Timeout	struct timeval(microseconds)	struct timespec(nanoseconds)	int in milliseconds
Timeout Modification	select() may modify the timeout struct	pselect() may modify the timeout struct	Timeout value not modified
Portability	Widely portable	Less portable (POSIX.1-2001)	Widely portable
Scalability	Poor for high fd numbers	Same as select()	Better scalability than select()
Error Values(errno)	EBADF, EINTR, EINVAL	Same as select()	EBADF, EINTR, ENOMEM, EINVAL

Summary of comparison

- Use **poll()** for better scalability and cleaner code when handling many file descriptors.
- Use **pselect()** if we need to **atomically block/unblock signals** during the wait (e.g., avoid race conditions).
- Avoid **select()** for large-scale applications due to its limitations with FD_SETSIZE and fd_set.

Concurrent Server Design

Overview of process and thread

- **Process** and **thread** are fundamental concepts in operating systems, representing units of execution. Understanding their differences, relationships, and uses is essential in systems programming and OS design.

Process

- A *process* is an independent program in execution. It has its own:
  - *Address space* (memory), *Code*, *data*, *heap*, and *stack*, *Open file descriptors*, *Execution context* (registers, program counter)
- Processes are managed by the operating system and are isolated from each other, providing *security and stability*.

Key Characteristics:

- Created using *fork(), exec()* in Unix/Linux.
- Switching between processes requires a *context switch*, which is relatively expensive.
- A process may contain one or more threads.

Overview of process and thread

Thread

- A *thread* (also called a *lightweight process*) is the smallest unit of CPU execution. A process can have *multiple threads*, which:
  - Share the *same address space*
  - Share *code*, *data*, *heap*, and *file descriptors*
  - Have their own *stack*, *program counter*, and *registers*
- Threads allow *concurrent execution* within a single process.

Key Characteristics

- Created using APIs like *pthread_create()* in POSIX systems or *std::thread* in C++.
- Faster context switching compared to processes.
- Ideal for *multitasking* within the same application (e.g., web server handling multiple clients).

Comparison: Process vs Thread

Feature	Process	Thread
Memory Space	Separate for each process	Shared within a process
Creation Time	Slower (fork(), exec())	Faster (pthread_create())
Context Switching	Expensive (due to full state switch)	Lightweight and faster
Communication	Through IPC (pipes, sockets, etc.)	Shared memory (easier but risky)
Failure Isolation	Safer — process crash doesn't affect others	Less safe — thread crash affects the whole process
Use Case	Independent apps (e.g., browser, editor)	Concurrent tasks (e.g., I/O, GUI + logic)

Use Cases:

- **Processes**: Isolated apps, security-sensitive operations, separate services.
- **Threads**: Parallel tasks in the same application, I/O-bound or compute-bound tasks, performance optimization.

Fork and Exec Functions

- ***fork()*** is called once but it returns twice.
- The creation of a new process is done using the *fork()* system call.
- A new program is run using the *exec(l,lp,le,v,vp)* family of system calls.
- These are two separate functions which may be used independently.
- A call to *fork()* will create a completely separate sub-process which will be exactly the same as the parent.
- The process that initiates the call to *fork* is called the *parent process.*
- The new process created by *fork* is called the *child process*.
- The child gets a copy of the parent's text and memory space.
- They do not share the same memory .

Fork return values

- **fork()** system call returns an integer to both the parent and child processes:
- **-1** this indicates an **error** with no child process created.
- A value of **zero** indicates that the child process code is being executed.
- Positive integers represent the **child's process identifier** (PID) and the code being executed is in the parent's process.

```
 if ( (pid = fork()) == 0)
  printf("I am the child\n");
       else
       printf("I am the parent\n");
```

The exec() System Call

- Calling one of the **exec()** family will terminate the currently running program and starts executing a new one which is specified in the parameters of exec in the context of the existing process. The process id is not changed.

EXEC Family of Functions

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

- The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified:
  - if filename contains a slash, it is taken as a pathname.
  - otherwise the executable file is searched for in the directories specified by the PATH environment variable.

Simple Execlp Example

```
#include <sys/type.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0){ /* error occurred */
                fprintf(stderr, “Fork Failed”);
                exit(-1);
        }
        else if (pid == 0){ /* child process */
                execlp(“/bin/ls”,”ls”,NULL);
        }
        else { /* parent process */
                /* parent will wait for child to complete */
                wait(NULL);
                printf(“Child Complete”);
                exit(0);
        }
}
```

Fork() and exec()

- When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.
- The **fork-and-exec** mechanism switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, and environment variables.

Purpose of exec() functions

- When a process calls one of the **exec** functions that process is completely replaced by the new program.
- The new program starts execution from **main** function.
- The process does not change across an exec because a new process is not created.
- But this function replaces the current process with new program from disk.

Fork() and exec()

Problems of fork() function

- **Fork** is expensive. Because memory and all descriptors are duplicated in the child.
- **Inter process communication** is required to pass information between the **parent** and the **child** after the fork.
- A **descriptor** in the child process can affect a subsequent read or write by the parent.
- This **descriptor** copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

wait() and waitpid()

Wait() / Waitpid() :

- Either of **wait** or **waitpid** can be used to remove zombies.
- **wait** (and **waitpid** in it's blocking form) temporarily suspends the execution of a parent process while a child process is running.
- Once the **child** has finished, the **waiting parent** is restarted.

Declarations:

```
#include <sys/types.h>
#include <sys/wait.h>
 pid_t wait(int *statloc); /* returns process ID if OK, or -1 on error */
 pid_t waitpid(pid_t pid, int *statloc, int options); /*returns
      process ID :if OK,
      0   : if non-blocking option && no zombies around
      -1  : on error
```

- The *statloc* argument can be one of two values:
- *NULL pointer*: the argument is simply ignored
- *pointer to an integer*: when ***wait*** returns, the integer this describes will contain status information of the terminated process.

wait() and waitpid()

Wait()	**Waitpid()**
wait blocks the caller until a child process terminates	**waitpid** can be either **blocking** or **non-blocking**: • If **options** is 0, then it is **blocking** • If **options** is **WNOHANG**, then is it **non-blocking**
if more than one **child** is running then **wait()** returns the first time one of the parent's offspring exits	**waitpid** is more flexible: • If **pid == -1**, it waits for any child process. In this respect, **waitpid** is equivalent to wait • If **pid > 0**, it waits for the child whose process ID equals **pid** • If **pid == 0**, it waits for any child whose process group ID equals that of the calling process • If **pid < -1**, it waits for any child whose process group ID equals that absolute value of **pid**

Process Related function

Functions those who are return a **process ID** are:

- **getpid ()** : returns the current **process ID.**
- **getppid ()** : returns the parent **process ID** of the **calling process**.
- **getuid()** : returns the real **user ID** of the **calling process**.
- **geteuid()**: returns the effective **user ID** of the calling process.
  - effective user ID gives the process additional permissions during execution of "set-user-ID'' mode processes
- **getgid ()**: returns the **real group ID** of the calling process
- **getegid()**: returns the **effective group ID** of the calling process.

Process group

- A process group is a collection of one or more processes.
- Each process group has a unique **process ID**.
- A function **getpgrp()** returns the process **group id** of the calling process.
- Each process group have a **leader**.
- The leader is identified by having its **process group ID equal its process ID**.
- A process joins an existing process group or creates a new process group by calling **setpgid()**.

Concurrent Server

- When a client request can take longer to service, we do not want to tie up a single server (*Iterative/sequential*) with one client; we want to handle multiple clients at the same time. The server that handles multiple clients simultaneously is a concurrent server.
- A concurrent server uses processes, *threads*, or *non-blocking I/O* to serve multiple clients in parallel - improving responsiveness and scalability.

Key Goals of a Concurrent Server:

- Accept multiple client connections.
- Handle each client independently.
- Avoid blocking the entire server while waiting on I/O for one client
- Efficiently use system resources

Common Approaches to Building Concurrent Servers

Approach	Description	Tools/Functions Used
Forking (Process-based)	Spawn a new process for each client connection	fork(), waitpid(), signal()
Threading	Spawn a new thread for each client	pthread_create(), pthread_join()
Event-driven (I/O multiplexing)	Use a single process/thread to monitor many clients using non-blocking I/O	select(),poll(),epoll()(Linux)
Preforked/Thread Pool	Pre-create a pool of worker processes/threads to handle connections	fork()/pthread_create() in advance

Workflow of a Concurrent Server

1. *Socket Creation*: Create a listening socket using *socket()*.
2. *Binding*: Bind the socket to an IP and port using *bind()*.
3. *Listening*: Listen for incoming connections using *listen().*
4. *Accepting Clients*: Use *accept()* in a loop.
5. *Concurrency*: For each connection:
   1. Fork a child process, or
   2. Create a thread, or
   3. Register the socket in an event loop (e.g., *select()*).
6. Serve the client: Communicate via *read()/write()* or *recv()/send()*.
7. Cleanup: Close client sockets and terminate the worker/thread as needed.

Pros and Cons of Each Model

Model	Pros	Cons
Forking	Simple to implement, good isolation	High overhead, not scalable for many clients
Threading	Lower overhead than processes	Needs careful synchronization, shared memory
Event-driven	Scales well, uses fewer resources	More complex logic, harder to debug
Thread Pool	Predictable resource usage	Adds management overhead

Real-world Usage:

- Apache (prefork or worker MPM) – process/thread pool models
- Nginx – event-driven (epoll)
- Node.js – single-threaded event loop
- Chat servers, web servers, game servers - all use concurrency patterns

Outline of concurrent server (forking model)

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(…);
/* fill in sockaddr_in{} with server's well-know 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);
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on *connfd*, the connected socket) and the parent process waits for another connection (on *listenfd*, the listening socket).
- The parent closes the connected socket since the child handles the new client. The function doit does whatever is required to service the client.
- Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. However, the close of *connfd* by the parent (in the outline) doesn't terminate its connection with the client. This is because every file or socket has a reference count.
- The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket. In the outline, after socket returns, the file table entry associated with *listenfd* has a reference count of 1.
- After accept returns, the file table entry associated with *connfd* has a reference count of 1. But, after fork returns, both descriptors are shared between the parent and child, so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes *connfd*, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes *connfd*.

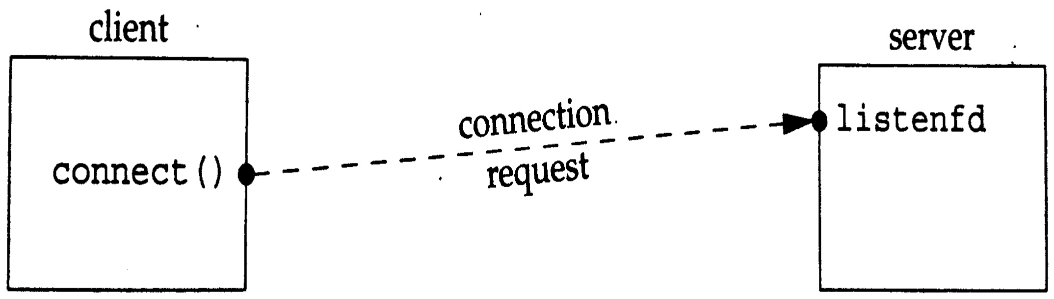


Figure 4.14 Status of client/server before call to accept returns.

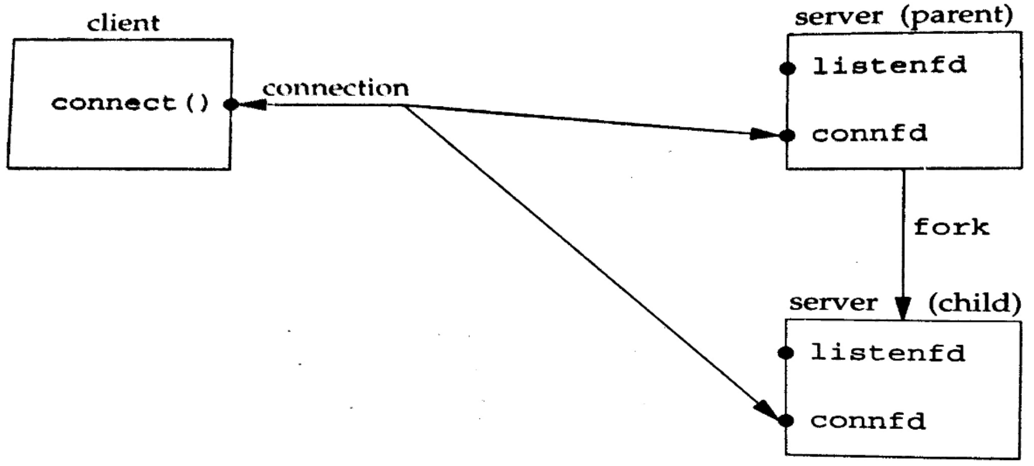


Figure 4.16 Status of client/server after fork returns.

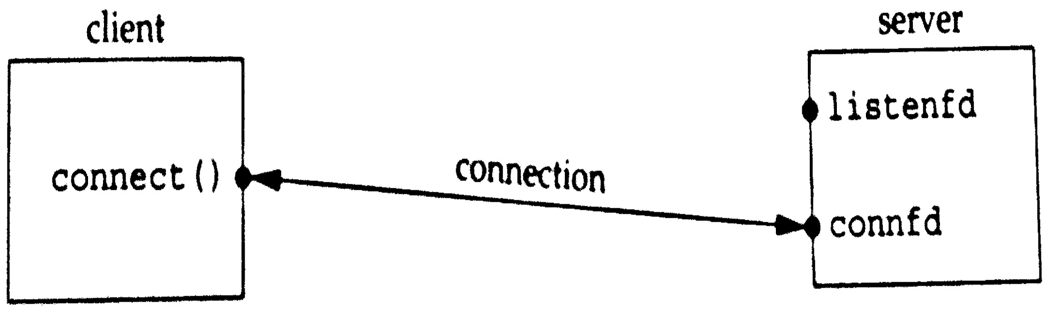


Figure 4.15 Status of client/server after return from accept.

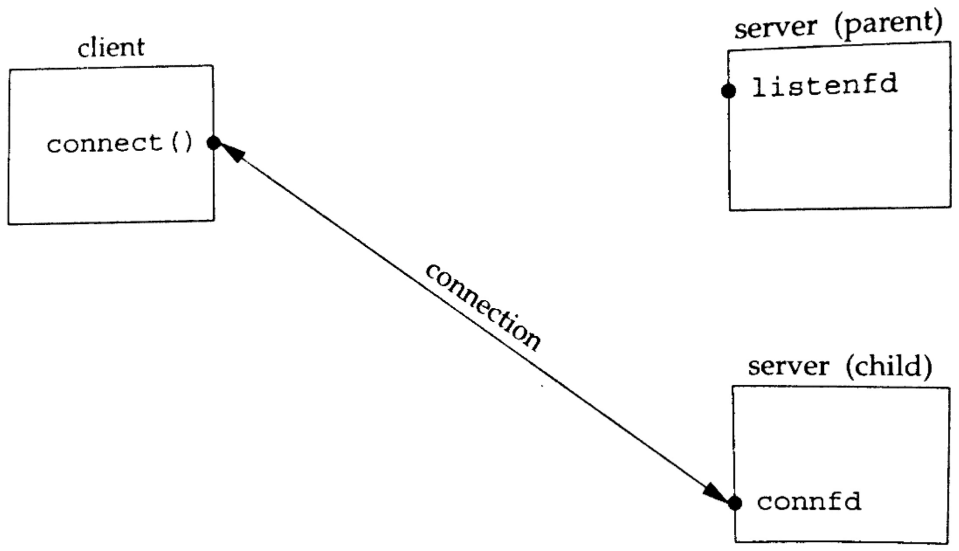


Figure 4.17 Status of client/server after parent and child close appropriate sockets.

Concurrent Server Using Thread Model

- A *thread-based concurrent server* creates a new thread for each incoming client connection. This allows the server to handle *multiple clients in parallel*, each served independently in its own thread of execution.
- Threads share the same memory space (unlike processes), so they can access shared resources easily but this also means *thread safety and synchronization* (e.g., using mutexes) are important if shared data is accessed.
- This model is efficient for I/O-bound applications (like web or chat servers), where most threads are blocked on I/O waiting for client input.

Advantages:

- Lower overhead than fork-based (no memory duplication).
- Easier communication between threads (shared memory).
- More scalable for moderate number of clients.

Disadvantages:

- Requires thread-safety handling (e.g., mutexes).
- Threads crashing can affect the whole process.

Outline of concurrent server (Multithreaded model)

```
#include <...> // Standard headers
#define PORT ...
#define MAX_CLIENTS ...
void *handle_client(void *arg) {
  read();
  write();
  close();
  pthread_exit();
}
int main() {
  socket();
  bind();
  listen();
  while (1) {
    accept();
    malloc();     // Allocate memory for client socket
    pthread_create(); // Create a new thread
    pthread_detach(); // Detach thread
  }
  close(); // Close server socket (if reached)
  return 0;
}
```

- The **main thread** creates a socket, binds it to a port, listens for incoming connections, and continuously accepts clients.
- For each client, it:
    - Allocates memory for the client socket
    - Spawns a **new thread** using pthread_create() to handle the client
    - Detaches the thread so resources are auto-reclaimed
- The handler thread performs read(), write()(to echo or process client data), and then closes the connection.
- Each client is handled concurrently in its own thread.

Example program: conserver_pthread.c

Concurrent Server Using I/O Multiplexing (using select)

- A select()-based server uses a single thread/process to monitor multiple client sockets simultaneously.
- It waits until one or more sockets become "ready" for I/O (read/write).
- This is more scalable than creating one thread per client and avoids the overhead of context switching.
- It is ideal for:
  - Moderate concurrency
  - Simple I/O-bound servers
- No need for multi-threading or synchronization

Outline of concurrent server (I/O Multiplexing)

```
#include <...> // Required headers
#define PORT ...
#define MAX_CLIENTS ...
int main() {
  socket();   //Create server socket
  bind();    //Bind to IP and port
  listen();   //Start listening
  FD_ZERO();   //Clear the fd_set
  FD_SET();   //Add server socket to the set
  select();  //Wait for activity on sockets
  while (1) {
    select(); // Monitor multiple sockets
    if (FD_ISSET(server_socket)) {
     accept(); //New connection
     FD_SET(); //Add new client socket to set
    }
```

```
 for (each client socket){
   if(FD_ISSET(client_socket)) {
     read(); // Handle client input
     write();// Respond if needed
     if (connection closed){
       close();// Close client socket
       FD_CLR();// Remove from fd_set
     }
   }
  }
}
close(); // Close server socket (if reached)
return 0;
}
```

- This is a single-threaded concurrent server that uses select() to monitor all sockets:
- When the listening socket is ready, a new client is accepted.
- When a client socket is ready, data is read/written.
- FD_SET, FD_CLR, and FD_ISSET manage sockets in the monitored set.

Example Program: conserver_select.c

Implementing broadcast and multicast communication

Broadcast, multicast and anycast

- **Unicast** sends packets from one sender to one receiver-the most common form.
- **Broadcast** delivers a packet to all hosts on a subnet simultaneously.
- **Multicast** sends a packet to a specific group of receivers (one-to-many).
- **Anycast** sends a packet to the nearest or most available receiver from a group offering the same service.

Communication	Destination	Scope	Use Case
Unicast	One specific host	One-to-one	HTTP, SSH, FTP
Broadcast	All on subnet	One-to-all	DHCP, ARP, service ping
Multicast	Group subscribers	One-to-many	IPTV, conferencing

Broadcast

- Sends packet to **all hosts on a subnet**
- Only works with UDP
- IP address: **Subnet broadcast address** (e.g., 192.168.1.255)
- Socket must be explicitly enabled for broadcast using SO_BROADCAST.
- TCP does not support broadcast (since it’s connection-oriented).

Limitations of Broadcast

Issue	Description
Network load	Every host receives the packet, even if unused.
Security	Can be abused for DoS (e.g., smurf attack).
No delivery guarantee	UDP is unreliable-packets may be dropped.
Not routable	Broadcasts do not cross routers.

Broadcast

Broadcast Programming Steps

1. Create UDP socket
2. Enable broadcast: *setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, ...)*
3. Set destination address: Broadcast IP
4. Send message using sendto()
5. Receiver binds to *INADDR_ANY*

Broadcast

Sender Code Outline

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
// Enable broadcast option
int broadcast = 1;

setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast, sizeof(broadcast));

// Set destination to broadcast address
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = inet_addr("192.168.1.255");

// Send broadcast message
sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&addr, sizeof(addr));
```

Broadcast

Receiver Code Outline

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = INADDR_ANY;

bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));

// Receive from any sender
recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);
```

Network Configuration Requirements

- Host must be connected to a subnet with broadcast enabled.
- Firewall and router must **allow UDP broadcast traffic** on the desired port.
- Proper subnet mask must be set to determine the correct broadcast address.

Example Program: broadcast_sender.c and broadcast_receiver.c

Multicast

- Multicast is a form of one-to-many communication.
- A sender sends a message to a multicast group address, and only subscribed receivers get the message.
- Efficient for group communication like IPTV, conferencing, and live data feeds.

Multicast IP Address Range (IPv4)

- *Class D addresses*: 224.0.0.0 – 239.255.255.255
- Common examples:
  - 224.0.0.1: All systems on subnet
  - 224.0.0.9: RIP routers
- These addresses are not assigned to specific hosts but to **groups**.

Characteristics of Multicast

Protocol	UDP only
Delivery	To subscribed members only
Group join/leave	Managed via IGMP (Internet Group Management Protocol)
TTL (Time To Live)	Controls how far packets can travel
Loopback	Sender can (optionally) receive its own message

Multicast

Programming Steps

Receiver:

1. Create UDP socket
2. Set SO_REUSEADDR (optional but useful for multiple receivers)
3. Bind to port and IP INADDR_ANY
4. Join multicast group with IP_ADD_MEMBERSHIP

Sender:

1. Create UDP socket
2. Set IP_MULTICAST_TTL if needed
3. Send packet to multicast group address + port

Note:

- Routers must support **IGMP** and allow multicast forwarding.
- On Linux, loopback interfaces may not forward multicast unless manually enabled.

Example Program: multicast_sender.c and multicast_receiver.c

Socket option

Socket Options

- **Socket options** are configuration parameters that modify the behavior of sockets at runtime.
- They allow fine control over:
  - Timeouts
  - Buffer sizes
  - Address reuse
  - Multicast behavior
  - TCP-level features (e.g., Nagle's algorithm)

- There are 3 ways to get and set options affecting sockets -
  - the *getsockopt* and *setsockopt* functions.
  - the *fcntl* function
  - the *ioctl* function

getsockopt() and setsockopt()

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

Both functions return 0 if OK else -1 on error.

- The *sockfd* must refer to an open socket descriptor.
- The *level* indicates whether the socket option is *general* or *protocol-specific* socket.
- The *optval* is a pointer to a variable from which the new value of the option is fetched by *setsockopt*, or into which the current value of the option is stored by *getsockopt*.
- The size of this variable is specified by the final argument, as a value for *setsockopt* and as a value-result for *getsockopt*.

getsockopt() and setsockopt()

- There are two basic types of options that can be queried by *getsockopt* or set by *setsockopt*: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values).
- When calling *getsockopt* for the flag options, ** optval* is an integer.
- The value returned in **optval* is *zero* if the option is disabled, or nonzero if the option is enabled.
- Similarly, *setsockopt* requires a *nonzero *optval* to turn the option on, and a *zero* value to turn the option off.
- For non-flag options, the option is used to pass a value of the specified datatype between the user process and the system.

Socket Options

- Two basic type of options -
  - Flags - binary options that enable or disable a feature.
  - Values - options that fetch and return specific values.
- Not supported by all implementations.
- Socket option fall into 4 main categories -
  - Generic socket options
    - SO_RCVBUF, SO_SNDBUF, SO_BROADCAST, etc.
  - IPv4
    - IP_TOS, IP_MULTICAST_IF, etc.
  - IPv6
    - IPv6_HOPLIMIT, IPv6_NEXTHOP, etc.
  - TCP
    - TCP_MAXSEG, TCP_KEEPALIVE, etc.

Common Socket Options

Level	Option	Purpose
SOL_SOCKET	SO_REUSEADDR	Allow reuse of local address
SOL_SOCKET	SO_RCVBUF, SO_SNDBUF	Set receive/send buffer size
SOL_SOCKET	SO_RCVTIMEO, SO_SNDTIMEO	Set timeout for recv/send
SOL_SOCKET	SO_BROADCAST	Enable sending broadcast datagrams
IPPROTO_IP	IP_TTL	Set IP time-to-live
IPPROTO_IP	IP_MULTICAST_TTL	TTL for multicast packets
IPPROTO_IP	IP_ADD_MEMBERSHIP	Join a multicast group
IPPROTO_TCP	TCP_NODELAY	Disable Nagle’s algorithm

Socket States

- Options have to be set or fetched depending on the state of a socket.
- Some socket options are inherited from a listening socket to the connected sockets on the server side.

  - E.g. **SO_RCVBUF** and **SO_SNDBUF**

    These options have to be set on the socket before calling *listen()* on the server side and before calling *connect()* on the client side.

Generic Socket Options

- **SO_BROADCAST**
  - Enables or disables the ability of a process to send broadcast messages.
  - It is supported only for datagram sockets.
  - Its default value is ***off***.
- **SO_ERROR**
  - *Pending Error* - When an error occurs on a socket, the kernel sets the ***so_error*** variable.
  - The process can be notified of the error in two ways -
    - If the process is blocked in ***select*** for either read or write, it returns with either or both conditions set.
    - If the process is using signal driven I/O, the ***SIGIO*** signal is generated for the process.

Generic Socket Options contd...

- **SO_KEEPALIVE**
  - Purpose of this option is to detect if the peer host crashes. The **SO_KEEPALIVE** option will detect half-open connections and terminate them.
  - If this option is set and no data has been exchanged for 2 hours, then TCP sends **keepalive** probe to the peer.
    - Peer responds with **ACK**. Another probe will be sent only after 2 hours of inactivity.
    - Peer responds with **RST** (has crashed and rebooted). Error is set to **ECONNRESET** and the socket is closed.
    - No response. 8 more probes are sent after which the socket's pending error is set to either **ETIMEDOUT** or **EHOSTUNREACH** and the socket is closed.

- Receive Low Water Mark -
  - Amount of **data** that must be in the socket receive buffer for a socket to become ready for *read.*
- Send Low Water Mark -
  - Amount of **space** that must be available in the socket send buffer for a socket to become ready for *write*.
- **SO_RCVLOWAT** and **SO_SNDLOWAT**
  - These options specify the receive low water mark and send low water mark for TCP and UDP sockets.

Generic Socket Options contd...

- **SO_RCVTIMEO** and **SO_SNDTIMEO**
  - These options place a timeout on socket receives and sends.
  - The timeout value is specified in a ***timeval*** structure.

```
struct timeval {
  long tv_sec ;
  long tv_usec ;
}
```

  - To disable a timeout, the values in the ***timeval*** structure are set to 0.

- **SO_REUSEADDR**
  - It allows a listening server to restart and bind its well known port even if previously established connections exist.
  - It allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.
  - It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
  - It allows completely duplicate bindings only for UDP sockets (broadcasting and multicasting).

Generic Socket Options Contd...

- **SO_LINGER**
  - Specifies how *close* operates for a connection-oriented protocol
  - The following structure is used:

```
struct linger {
  int l_onoff;
  int l_linger;
}
// l_onoff - 0=off; nonzero=on
// l_linger specifies seconds
```

Three scenarios:

1. If ***l_onoff*** is 0, ***close*** returns immediately. If there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The value of ***l_linger*** is ignored.
2. If ***l_onoff*** is **nonzero** and ***linger*** is 0, TCP aborts the connection when *close* is called. TCP discards data in the send buffer and sends **RST** to the peer.

Generic Socket Options Contd…

3. SO_LINGER (cont)

If ***l_onoff*** is nonzero and ***linger*** is nonzero, the kernel will linger when *close* is called.

- If there is any data in the send buffer, the process is put to sleep until either:
  - the data is sent and acknowledged
  - Or
  - the linger time expires (for a nonblocking socket the process will not wait for *close* to complete)
- When using this feature, the return value of *close* must be checked. If the linger time expires before the remaining data is send and acknowledged, close returns **EWOULDBLOCK** and any remaining data in the buffer is ignored.

Generic Socket Options Contd...

- **SO_SNDBUF/SO_RCVBUF**
  - Level: SOL_SOCKET
  - Get/Set supported
  - Non-flag option
  - Datatype of optval: int
  - Description: Send buffer size/Receive buffer size

- **TCP_NODELAY**
  - Level: IPPROTO_TCP
  - Get/Set supported
  - Flag option i.e. enable or disable Nagle algorithm
  - Description: Disable/Enable Nagle algorithm

Socket Options Summary

Socket Option	Level	Use Case
SO_REUSEADDR	SOL_SOCKET	Reuse port
SO_RCVBUF	SOL_SOCKET	Tune performance
SO_BROADCAST	SOL_SOCKET	Enable UDP broadcast
SO_RCVTIMEO	SOL_SOCKET	Set recv() timeout
TCP_NODELAY	IPPROTO_TCP	Disable Nagle (for real-time apps)
IP_TTL	IPPROTO_IP	Set packet TTL
IP_MULTICAST_TTL	IPPROTO_IP	Set multicast range
IP_ADD_MEMBERSHIP	IPPROTO_IP	Join multicast group

fcntl()function

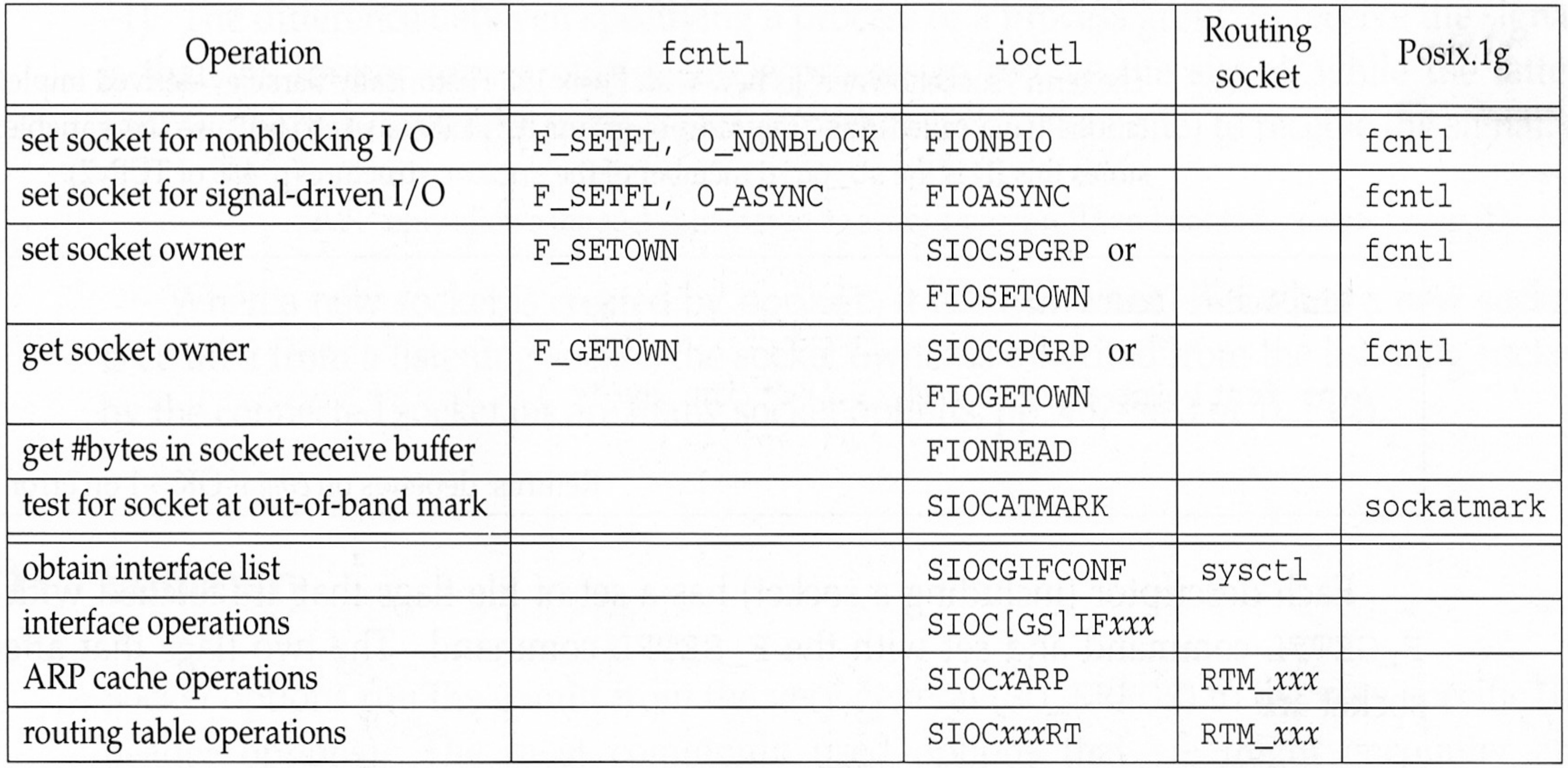
- *fcntl()* stands for “file control” and this function performs various descriptor control operations.
- The fcntl function provides the following features related to network programming.
  - *Non-blocking I/O* – We can set the *O_NONBLOCK* file status flag using the *F_SETFL* command to set a socket as non-blocking.
  - *Signal-driven I/O* – We can set the *O_ASYNC* file status flag using the *F_SETFL* command, which causes the SIGIO signal to be generated when the status of a socket changes.
  - The *F_SETOWN* command lets us set the socket owner (the process ID or process group ID) to receive the *SIGIO* and *SIGURG* signals. The former signal is generated when the signal-driven I/O is enabled for a socket and the latter signal is generated when new out-of-band data arrives for a socket. The *F_GETOWN* command returns the current owner of the socket.

fcntl()…

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

- Each descriptor has a set of file flags that is fetched with the *F_GETFL* command and set with the *F_SETFL* command. The two flags that affect a socket are
- O_NONBLOCK - non-blocking I/O
- O_ASYNC-signal-driven I/O
- Miscellaneous file control operations
  - Non-blocking I/O (O_NONBLOCK, F_SETFL)
  - Signal-driven I/O (O_ASYNC, F_SETFL)
  - Set socket owner (F_SETOWN)

fcntl and ioctl



Operation	fcntl	ioctl	Routing socket	Posix.1g
set socket for nonblocking I/O	`F_SETFL, O_NONBLOCK`	`FIONBIO`		`fcntl`
set socket for signal-driven I/O	`F_SETFL, O_ASYNC`	`FIOASYNC`		`fcntl`
set socket owner	`F_SETOWN`	`SIOCSPGRP` or `FIOSETOWN`		`fcntl`
get socket owner	`F_GETOWN`	`SIOCGPGRP` or `FIOGETOWN`		`fcntl`
get #bytes in socket receive buffer		`FIONREAD`		
test for socket at out-of-band mark		`SIOCATMARK`		`sockatmark`
obtain interface list		`SIOCGIFCONF`	`sysctl`	
interface operations		`SIOC[GS]IF`*xxx*		
ARP cache operations		`SIOC`*x*`ARP`	`RTM_`*xxx*	
routing table operations		`SIOC`*xxx*`RT`	`RTM_`*xxx*	

Figure 7.15 Summary of `fcntl`, `ioctl`, and routing socket operations.

fcntl() function

- *fcntl* provides the following features related to network programming
  - **Nonblocking I/O** (be aware of error-handling in the following code)

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

  - **Signal driven I/O**

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_ASYNC;
fcntl(fd, F_SETFL, flags);
```

  - Set socket owner to receive SIGIO signals

```
fcntl(fd, F_SETOWN, getpid());
```

fcntl() function

- The signals *SIGIO* and *SIGURG* are generated for a socket only if the socket has been assigned an owner with the *F_SETOWN* command. The integer arg value for the *F_SETOWN* command can be either positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal.
- The *F_GETOWN* command returns the socket owner as the return value from the fcntl function, either the process ID (a positive return value) or the process group ID (a negative value other than -1).
- The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group to receive the signal.

ioctl operations

- The common use of *ioctl* by network programs (typically servers) is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether interface supports broadcasting, whether the interface supports multicasting, and so on.

- **Ioctl() Function**
- This function affects an open file referenced by the **fd** argument.

```
#include <unistd.h>
 int ioctl(int fd, int request, …/* void *arg */);
 Returns: 0 if OK, -1 on error
```

- The third argument is always a pointer, but the type of pointer depends on the request.

Example (Convert to Non blocking)

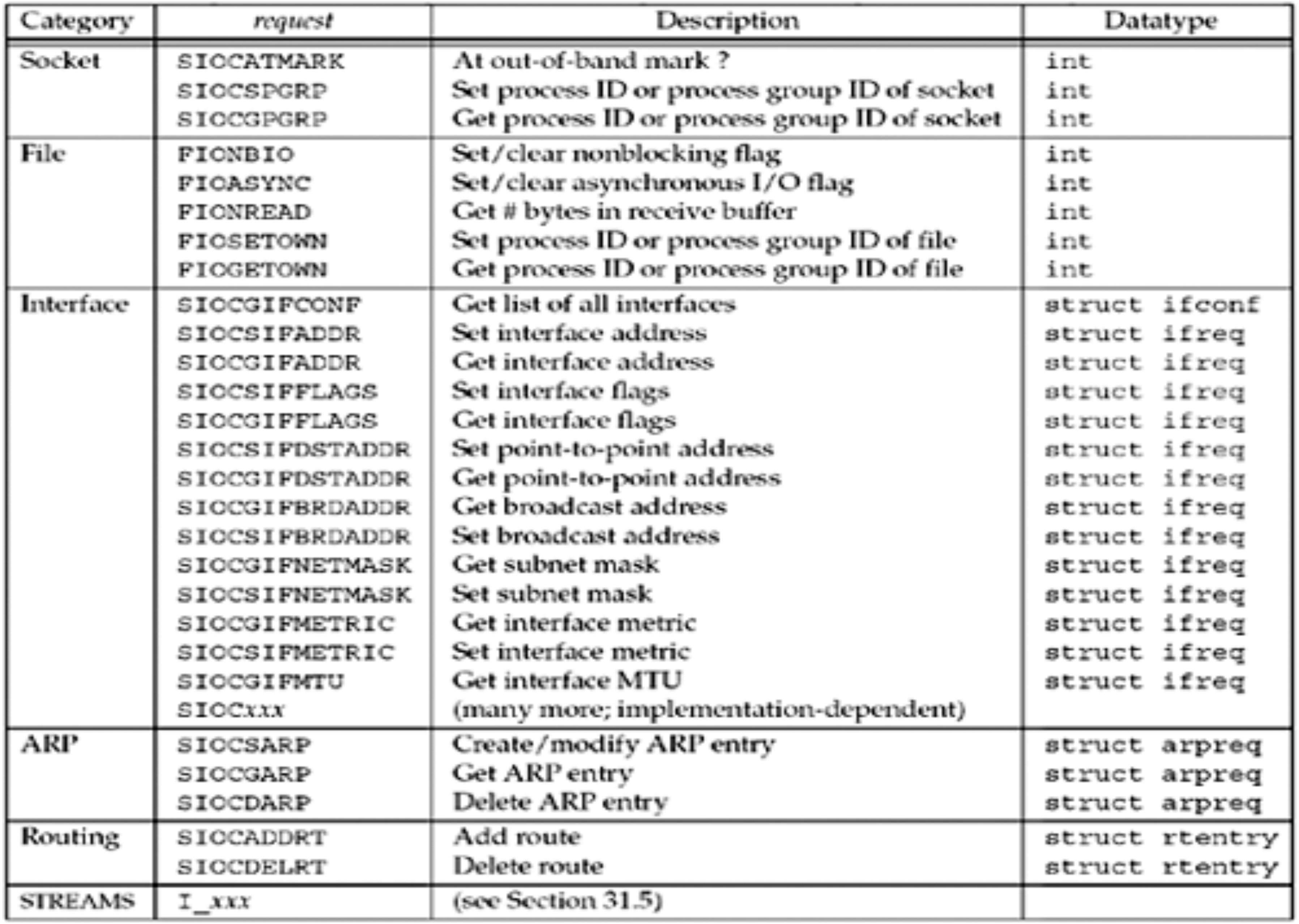
```
int flags = fcntl(fd, F_GETFL, 0);
int status = ioctl(fd, FIONBIO, &flags);
if (status < 0) {
 perror("ioctl");
 exit(EXIT_FAILURE);
}
```

ioctl() Function

- We can divide the requests related to networking into six categories.

1. Socket operations
2. File operations
3. Interface operations
4. ARP cache operations
5. Routing table operations
6. STREAMS system

Note that not only do some of the **ioctl** operations overlap some of the **fcntl** operations (e.g. setting a socket to non-blocking), but there are also some operations that can be specified more than one way using **ioctl** (e.g., setting the process group ownership of a socket).



Category	*request*	Description	Datatype
Socket	`SIOCATMARK`	At out-of-band mark ?	`int`
	`SIOCSPGRP`	Set process ID or process group ID of socket	`int`
	`SIOCGPGRP`	Get process ID or process group ID of socket	`int`
File	`FIONBIO`	Set/clear nonblocking flag	`int`
	`FIOASYNC`	Set/clear asynchronous I/O flag	`int`
	`FIONREAD`	Get # bytes in receive buffer	`int`
	`FIOSETOWN`	Set process ID or process group ID of file	`int`
	`FIOGETOWN`	Get process ID or process group ID of file	`int`
Interface	`SIOCGIFCONF`	Get list of all interfaces	`struct ifconf`
	`SIOCSIFADDR`	Set interface address	`struct ifreq`
	`SIOCGIFADDR`	Get interface address	`struct ifreq`
	`SIOCSIFFLAGS`	Set interface flags	`struct ifreq`
	`SIOCGIFFLAGS`	Get interface flags	`struct ifreq`
	`SIOCSIFDSTADDR`	Set point-to-point address	`struct ifreq`
	`SIOCGIFDSTADDR`	Get point-to-point address	`struct ifreq`
	`SIOCGIFBRDADDR`	Get broadcast address	`struct ifreq`
	`SIOCSIFBRDADDR`	Set broadcast address	`struct ifreq`
	`SIOCGIFNETMASK`	Get subnet mask	`struct ifreq`
	`SIOCSIFNETMASK`	Set subnet mask	`struct ifreq`
	`SIOCGIFMETRIC`	Get interface metric	`struct ifreq`
	`SIOCSIFMETRIC`	Set interface metric	`struct ifreq`
	`SIOCGIFMTU`	Get interface MTU	`struct ifreq`
	`SIOC`*xxx*	(many more; implementation-dependent)	
ARP	`SIOCSARP`	Create/modify ARP entry	`struct arpreq`
	`SIOCGARP`	Get ARP entry	`struct arpreq`
	`SIOCDARP`	Delete ARP entry	`struct arpreq`
Routing	`SIOCADDRT`	Add route	`struct rtentry`
	`SIOCDELRT`	Delete route	`struct rtentry`
STREAMS	`I_`*xxx*	(see Section 31.5)	

Socket operations

- Three **ioctl** requests are explicitly used for sockets. All three require that the third argument to **ioctl** be a pointer to an integer.
  - **SIOCATMARK**: Return through the integer pointed to by the third argument a non-zero value if the socket's read pointer is currently at the out-of-band mark, or a zero value if the read pointer is not at the out-of-band mark.
  - **SIOCGPGRP**: Return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive **SIGIO** or SIGURG signal for this socket. This request is identical to an fcntl of **F_GETOWN**, note that POSIX standardizes the fcntl.
  - **SIOCSPGRP:** Set either the process ID or process group ID to receive the **SIGIO** or **SIGURG** signal for this socket from the integer pointed to by the third argument. This request is identical to an fnctl of **F_SETOWN**, note that POSIX standardizes the **fcntl**.

File Operations

- The next group of requests begin with FIO and may apply to certain types of files, in addition to sockets.

FIONBIO	The nonblocking flag for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This request has the same effect as the **O_NONBLOCK** file status flag, which can be set and cleared with the **F_SETFL** command to the fcntl function.
FIOASYNC	The flag that governs the receipt of asynchronous I/O signals (SIGIO) for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This flag has the same effect as the O_ASYNC file status flag, which can be set and cleared with the F_SETFL command to the fcntl function.
FIONREAD	Return in the integer pointed to by the third argument to ioctl the number of bytes currently in the socket receive buffer. This feature also works for files, pipes, and terminals.
FIOSETOWN	Equivalent to SIOCSPGRP for a socket.
FIOGETOWN	Equivalent to SIOCGPGRP for a socket.

Interface Operations

- The **SIOCGIFCONF** request returns the name and a socket address structure for each interface that is configured. Many of requests use a socket address structure to specify or return an IP address or address mask with the application.

SIOCGIFADDR	Return the unicast address in the ifr_addr member.
SIOCSIFADDR	Set the interface address from the ifr_addr member. The initialization function for the interface is also called.
SIOCGIFFLAGS	Return the interface flags in the ifr_flags member. The names of the various flags are IFF_*xxx* and are defined by including the <net/if.h> header.
SIOCSIFFLAGS	Set the interface flags from the ifr_flags member.
SIOCGIFDSTADDR	Return the point-to-point address in the ifr_dstaddr member.
SIOCSIFDSTADDR	Set the point-to-point address from the ifr_dstaddr member.
SIOCGIFBRDADDR	Return the broadcast address in the ifr_broadaddr member. The application must first fetch the interface flags and then issue the correct request: SIOCGIFBRDADDR for a broadcast interface or SIOCGIFDSTADDR for a point-to-point interface.
SIOCSIFBRDADDR	Set the broadcast address from the ifr_broadaddr member.
SIOCGIFNETMASK	Return the subnet mask in the ifr_addr member.
SIOCSIFNETMASK	Set the subnet mask from the ifr_addr member.
SIOCGIFMETRIC	Return the interface metric in the ifr_metric member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon routed. The interface metric is added to the hop count (to make an interface less favorable).
SIOCSIFMETRIC	Set the interface routing metric from the ifr_metric member.

ARP Cache Operations

- On some systems, the ARP cache is also manipulated with the **ioctl** function. Systems that use routing sockets usually use routing sockets instead of **ioctl** to access the ARP cache. These requests use an **arpreq** structure, shown below and defined by including the <net/if_arp.h> header.

```
<net/if_arp.h>
struct arpreq {
struct sockaddr arp_pa; /* protocol address */
struct sockaddr arp_ha; /* hardware address */
int arp_flags; /* flags */
};
#define ATF_INUSE 0x01 /* entry in use */
#define ATF_COM 0x02 /* completed entry (hardware addr valid) */
#define ATF_PERM 0x04 /* permanent entry */
#define ATF_PUBL 0x08 /* published entry (respond for other host) */
```

ARP Cache Operations

- The third argument to ioctl must point to one of these structures. The following three *requests* are supported

SIOCSARP	Add a new entry to the ARP cache or modify an existing entry. arp_pa is an Internet socket address structure containing the IP address, and arp_ha is a generic socket address structure with sa_family set to AF_UNSPEC and sa_data containing the hardware address (e.g., the 6-byte Ethernet address). The two flags, ATF_PERM and ATF_PUBL, can be specified by the application. The other two flags, ATF_INUSE and ATF_COM, are set by the kernel.
SIOCDARP	Delete an entry from the ARP cache. The caller specifies the Internet address for the entry to be deleted.
SIOCGARP	Get an entry from the ARP cache. The caller specifies the Internet address, and the corresponding Ethernet address is returned along with the flags.

Routing Table Operations

- On some systems, two **ioctl** requests are provided to operate on the routing table. These two requests require that the third argument to **ioctl** be a pointer to an **rtentry** structure, which is defined by including the <net/route.h> header. These requests are normally issued by the route program. Only the superuser can issue these requests. On systems with routing sockets, these requests use routing sockets instead of ioctl.

SIOCADDRT	Add an entry to the routing table.
SIOCDELRT	Delete an entry from the routing table.

- There is no way with **ioctl** to list all the entries in the routing table. This operation is usually performed by the **netstat** program when invoked with the -r flag. This program obtains the routing table by reading the kernel's memory (/dev/kmem).

IOCTL summary

The ioctl commands that are used in network programs can be divided into six categories:

1. Socket operations (Are we at the out-of-band mark?)
2. File operations (set or clear the nonblocking flag)
3. Interface operations (return interface list, obtain broadcast address)
4. ARP table operations (create, modify, get, delete)
5. Routing table operations (add or delete)
6. STREAMS system

Logging in Unix

syslogd Daemon

- Unix systems normally start a daemon named **syslogd** from one of the system initialization scripts, and it runs as long as the system is up. The **syslogd** perform the following actions on startup.
  - The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log messages that the daemon can receive.
  - A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
  - A UDP socket is created and bound to port 514 (the syslog service).
  - The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.
- The **syslogd** daemon runs in an infinite loop that calls **select**, waiting for any one of its three descriptors (last three of above bullets) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGUP signal, it reads its configuration file

syslog function

- Since a daemon does not have a controlling terminal, it cannot just **fprintf** to **stderr**. The common technique for logging messages from a daemon is to call the syslog function.

```
#include <syslog.h>
 void syslog(int priority, const char * message,…);
```

- The priority argument is a combination of a level and a facility. The message is like a format string to printf, with addition of a %m specification, which is replaced with error message corresponding to the current value of **errno**.
- For example, the following call could be issued by a daemon when a call to the rename function unexpectedly fails:

```
syslog(LOG_INFO| LOG_LOCAL2, “rename (%s, %s): %m”, file1, file2);
```

Common Implementations

Daemon Name	Description
syslogd	Traditional BSD syslog daemon. Lightweight and basic.
rsyslogd	Default on most Linux distros now. Adds filtering, remote logging, and database output.
syslog-ng	Advanced syslog daemon with better performance and security features.
journald	Systemd-based logging (used in modern Linux systems), replaces traditional syslog in many distros.

Common Syslog Log Files

File Path	Purpose
/var/log/syslog	General system messages
/var/log/messages	General log (on some distros)
/var/log/auth.log	Authentication events
/var/log/daemon.log	Logs from daemons
/var/log/kern.log	Kernel messages

Testing syslog

```
#include <syslog.h>

int main() {
 openlog("myapp", LOG_PID | LOG_CONS, LOG_DAEMON);
 syslog(LOG_INFO, "This is a test log from my daemon.");
 closelog();
 return 0;
}
```

Log Priorities (Severity Levels)

Priority Level	Constant for syslog()	Constant for os_log()	Description
Emergency	LOG_EMERG	OS_LOG_TYPE_FAULT	System is unusable. Highest severity. Alerts for catastrophic system failures or panic.
Alert	LOG_ALERT	OS_LOG_TYPE_ERROR	Immediate action required. Severe system issues, like disk failure, that require immediate attention.
Critical	LOG_CRIT	OS_LOG_TYPE_ERROR	Critical conditions like hardware failures, but not as urgent as "alert."
Error	LOG_ERR	OS_LOG_TYPE_ERROR	Non-critical issues like application errors, system errors, or recoverable failures.
Warning	LOG_WARNING	OS_LOG_TYPE_INFO	Conditions that might indicate potential problems (e.g., low disk space, deprecated API usage).
Notice	LOG_NOTICE	OS_LOG_TYPE_INFO	Normal but significant events that are important to log (e.g., routine logins, background tasks).
Informational	LOG_INFO	OS_LOG_TYPE_INFO	Informational messages that track the normal operation of the system (e.g., service started).
Debug	LOG_DEBUG	OS_LOG_TYPE_DEBUG	Detailed debug-level information for troubleshooting purposes (e.g., verbose output for developers).

Introduction to P2P programming

P2P programming

- What is Peer-to-Peer (P2P) Programming?
  - Peer-to-Peer (P2P) is a decentralized network model in which each participant (peer) can act as both a client and a server.
  - Unlike client-server, there is no central server.
  - Each node shares and consumes resources (files, messages, etc.).
  - Common in file sharing (BitTorrent), communication tools (Skype), blockchain, etc.

Key Characteristics of P2P

Feature	Description
Decentralization	No single point of failure
Scalability	More peers → more capacity
Resource sharing	Bandwidth, storage, and data shared
Fault tolerance	Network can continue despite node failures

Flow in a P2P Application

Peer A		Peer B
socket()		socket()
bind()		bind()
listen()		listen()
connect()	←connect()..	connect()
send()/recv()	←exchange()→	send()/recv()
close()		close()

Challenges in P2P Programming

- **Peer Discovery**: How does one peer find another? (e.g., bootstrap servers)
- **NAT Traversal**: Many peers are behind routers; connecting across networks may require STUN/TURN.
- **STUN** stands for **Session Traversal Utilities for NAT**. It is a **network protocol** that allows a device behind a NAT (Network Address Translation) or firewall to discover its **public IP address and port** as seen by the outside world.
  - The client (behind NAT) sends a request to a **STUN server** on the public internet.
  - The STUN server replies with:
    - The **public IP and port** it sees for the request.
  - The client now knows:
    - "This is how I appear to the outside world."
  - TURN is used when:
    - STUN fails to establish a direct connection (e.g., due to **symmetric NAT**, **firewalls**, or **corporate networks**).
    - Peers can't see or connect to each other directly.
    - A fallback relay is needed to route the data.
    - Without TURN, applications like video calls, chat, or file sharing would **fail in restrictive networks**.
- **Security**: No central control, so encryption, authentication, and trust are vital.
- **Concurrency**: A peer may serve and connect to multiple peers at once → use threads or select().

Overview Network Security Programming

Network Security Programming

- **Network Security Programming** involves writing software that ensures **confidentiality, integrity, and availability** of data transmitted over computer networks. It implements mechanisms to **detect**, **prevent**, and **respond** to unauthorized access or attacks.

Goals of Network Security

Goal	Description
Confidentiality	Ensure that only authorized parties can read data
Integrity	Prevent unauthorized data modification
Availability	Ensure services are accessible and usable
Authentication	Verify identity of communicating parties
Non-repudiation	Prevent denial of actions (e.g., sending messages)

Common Threats in Network Communication

- *Eavesdropping (Sniffing)* – Reading data in transit
- *Man-in-the-Middle (MitM)* – Intercepting/modifying communication
- *IP Spoofing* – Pretending to be another host
- *Denial-of-Service (DoS)* – Overloading a service to make it unavailable
- *Replay Attacks* – Re-sending captured data

Network Security Programming

Key Concepts & Techniques

Concept	Description
Encryption	Secure data in transit using ciphers
Digital Signature	Ensure data integrity and authenticity
TLS/SSL	Encrypts TCP connections (e.g., HTTPS)
VPNs	Secure tunneling of network traffic
Firewalls	Filter network traffic
Hashing	Generate fingerprint for data integrity

Programming Tools & APIs

Library/Tool	Usage
OpenSSL	Encryption, TLS, key generation
GnuTLS	Secure network communication
libpcap	Packet capture and monitoring
iptables (Linux)	Firewall scripting
Socket API	Core for implementing secure protocols

Secure Programming Workflow

1. Create TCP/UDP socket (socket())
2. Setup encryption context (OpenSSL, TLS)
3. Authenticate peers (certificates, keys)
4. Encrypt and transmit data (SSL_write)
5. Decrypt and read data (SSL_read)
6. Handle errors and clean up (SSL_free, close)

Securing by Hostname or Domain Name

- Hostname verification ensures that a **server's certificate** matches the expected domain.
- Prevents **man-in-the-middle (MitM)** attacks where an attacker may impersonate the server.
- **Common Name (CN)** or **Subject Alternative Name (SAN)** in certificates must match the requested domain.
- Why is Hostname Verification Necessary?
  - **Risk of Impersonation:** Without verification, an attacker could impersonate a trusted server.
  - **Security Layer:** Prevents redirecting communication to a malicious server.
    - **Example:** Client requests https://securebank.com, but an attacker replies with a valid certificate for attacker.com.
  - **Outcome:** The client would unknowingly trust the malicious server.

Securing by Hostname or Domain Name

- **Header:** How Does Hostname Verification Work?
  - **Client sends request**: https://example.com
  - **Server returns certificate**: Contains CN/SAN fields.
  - **Client compares**: Hostname against CN or SAN in the certificate.
  - **Connection succeeds**: If they match.
  - **Connection fails**: If they do not match (MitM prevention).

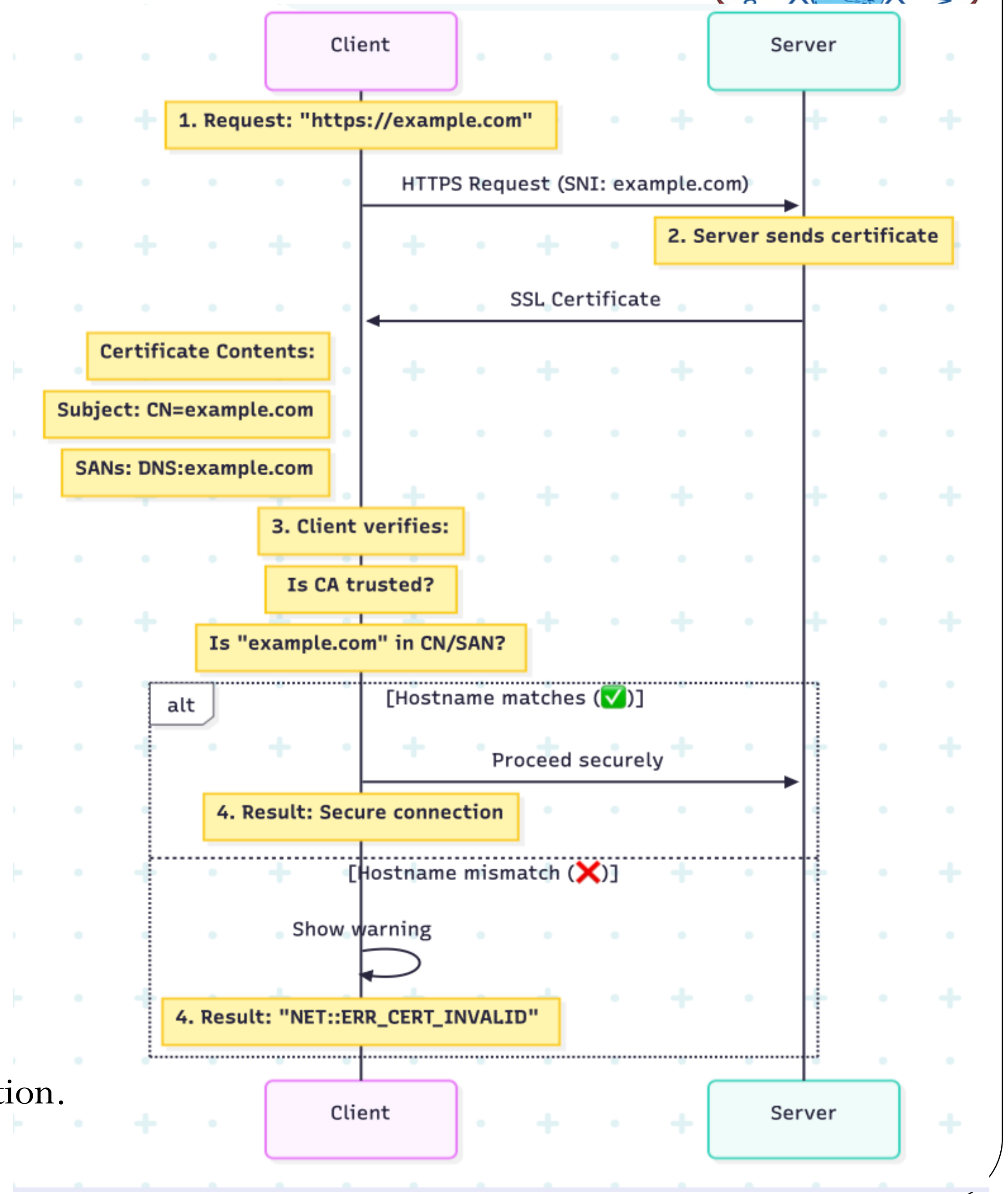


Diagram: The flow of the client-server communication with certificate verification.

Generated by: *https://www.mermaidchart.com/*

Identification by IP Number

- **Identification by IP number** refers to verifying the identity of a connecting client or server based on its IP address. This technique can be used as part of **access control** or **connection filtering**, especially in **server-side security** implementations.
- When to Use IP-Based Identification?
  - We control both client and server IP spaces
  - The network is private or internal
  - Simplicity is more important than flexibility
- Do not use it when:
  - Clients move between networks
  - Strong security/authentication is required
- In a real server, we would extract the client IP using *getpeername()* and convert it to string using *inet_ntop().*
- *Program: **ip_filter_server.c***

What is a Wrapper Program?

- A **wrapper program**:
  - Runs **before** the actual service,
  - Performs checks like:
    - **IP address filtering**
    - **User ID / privilege checks**
    - **Time of day**
  - If all checks pass, it launches the target service using exec*() family of system calls.
  - Program: *secure_wrapper.c*

Policy Features Can Add (Homework)

- Time of day: is_allowed_time()
- User identity: getuid(), getpwuid()
- Source IP (if used for network access): Check getpeername() if this is a socket-based wrapper
- Log every access: Log to a file using syslog() or fprintf()
- Limit usage frequency: Store access times in a file or memory

End of Chapter 3