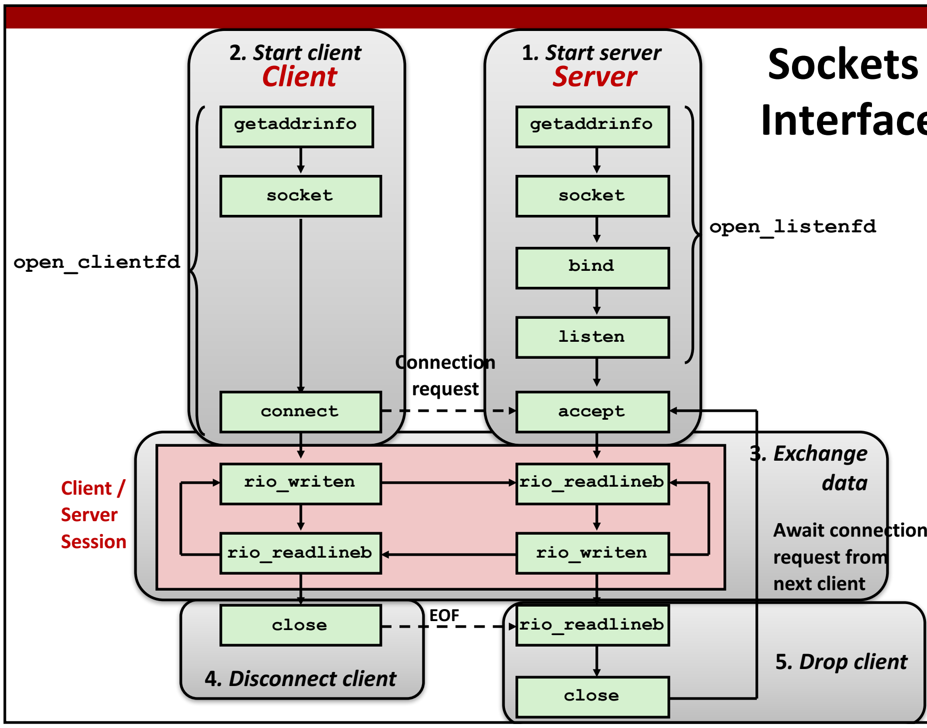
Network Programming: Part II

CS 475

1

Sockets Interface



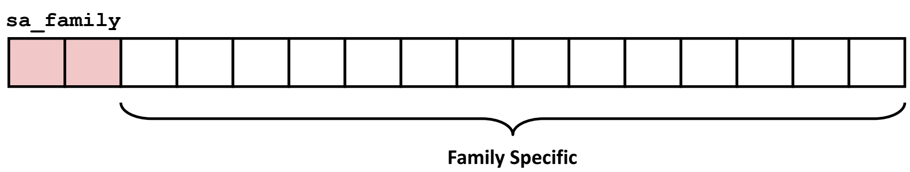
2

Socket Address Structures

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`**
  - Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed
  - For casting convenience, we adopt the Stevens convention:

    `typedef struct sockaddr SA;`

```
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data.  */
};
```

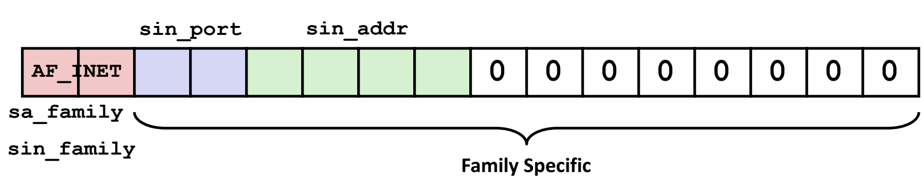


3

Socket Address Structures

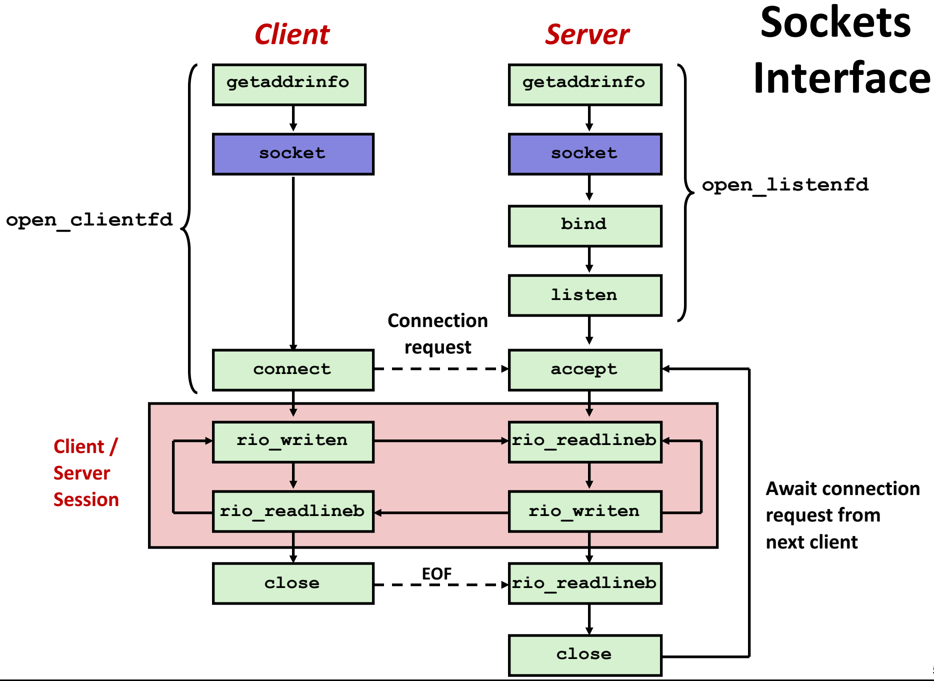
- **Internet-specific socket address:**
  - Must cast (`struct sockaddr_in *`) to (`struct sockaddr *`) for functions that take socket address arguments.

```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



4

Sockets Interface



Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a *socket descriptor*:

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

- Example:

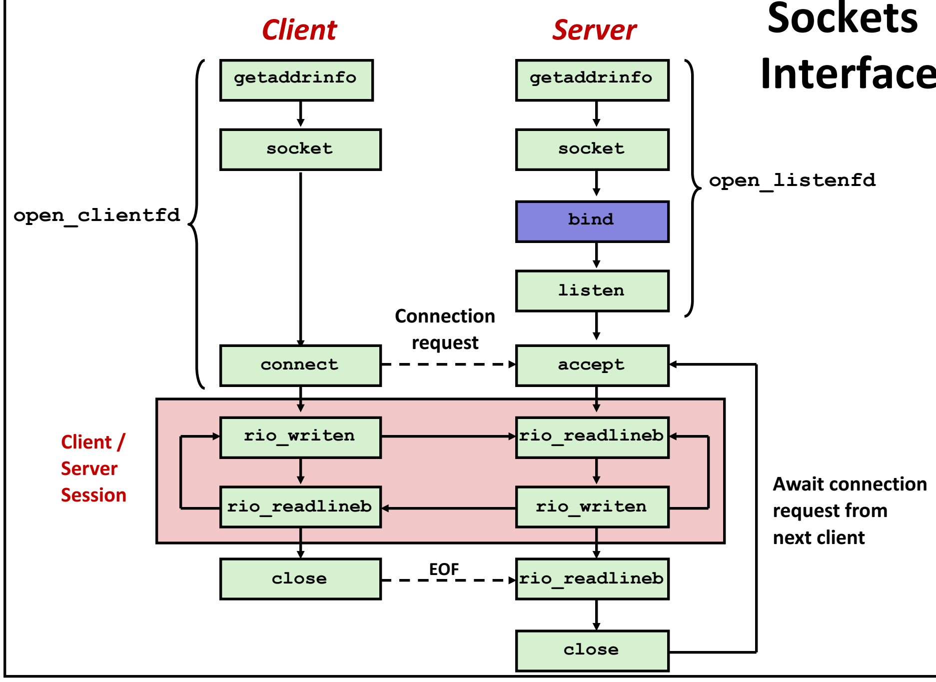
```
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

Indicates that the socket will be the end point of a connection

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface



Sockets Interface: `bind`

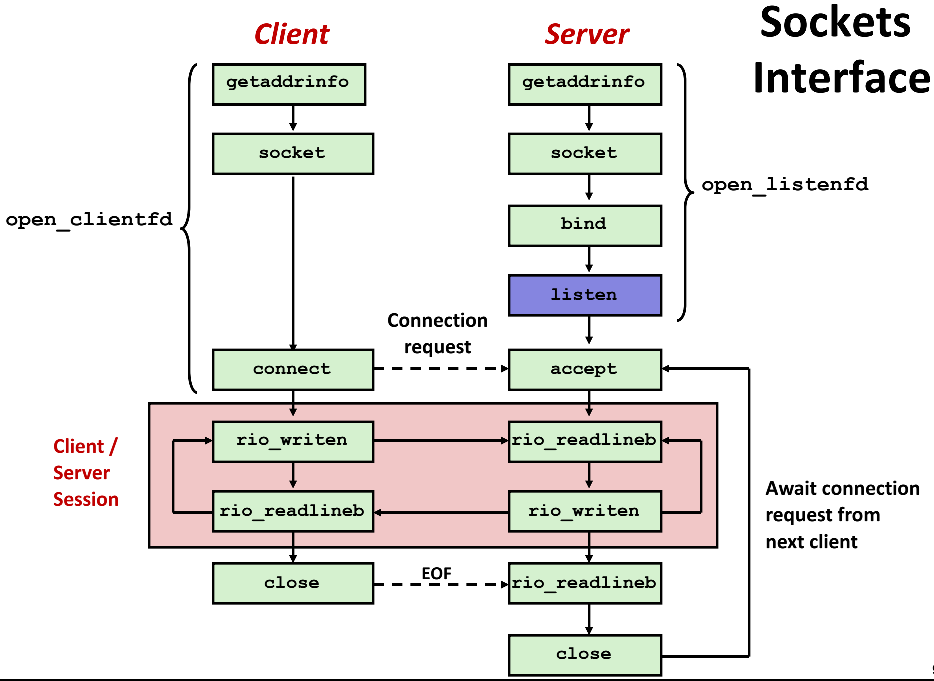
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr` .

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

Sockets Interface



9

Sockets Interface: `listen`

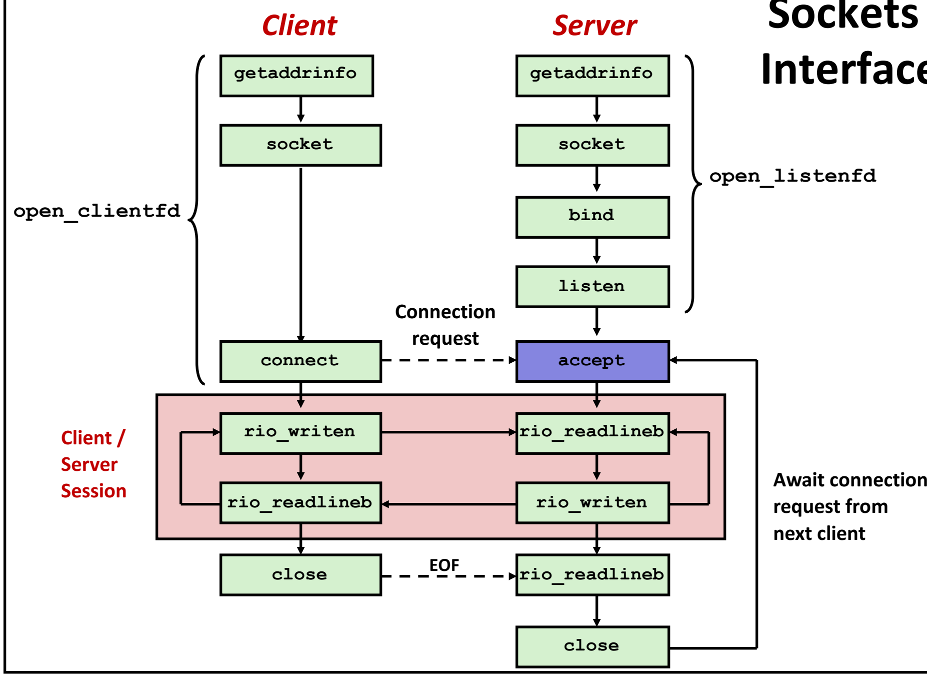
- By default, kernel assumes that descriptor from socket function is an ***active socket*** that will be on the client end of a connection.
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

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

- Converts `sockfd` from an active socket to a ***listening socket*** that can accept connection requests from clients.
- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

10

Sockets Interface

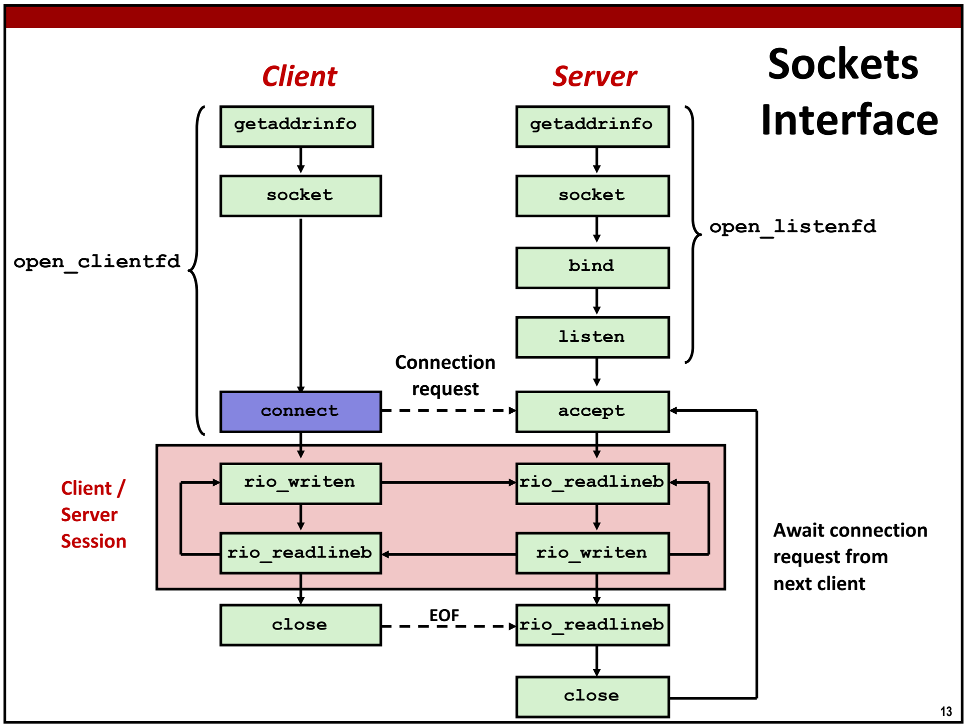


Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.
- Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.



Sockets Interface: `connect`

- **A client establishes a connection with a server by calling connect:**

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

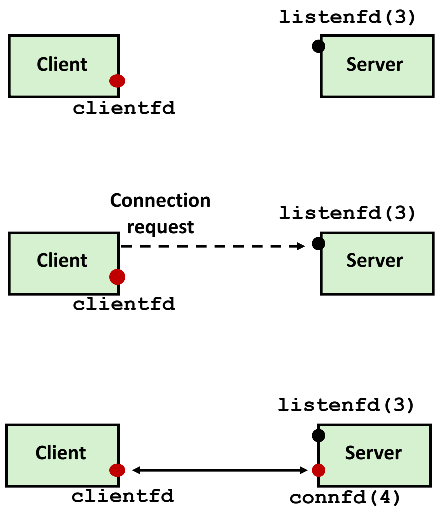
- **Attempts to establish a connection with server at socket address `addr`**
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

    `(x:y, addr.sin_addr:addr.sin_port)`
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

14

`accept` Illustrated



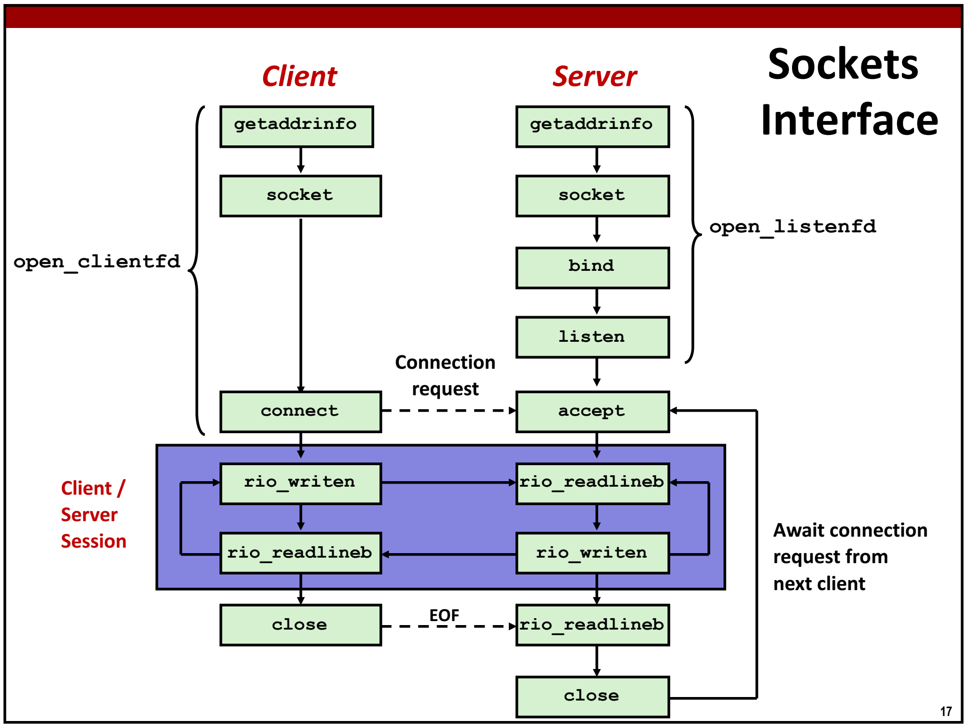
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

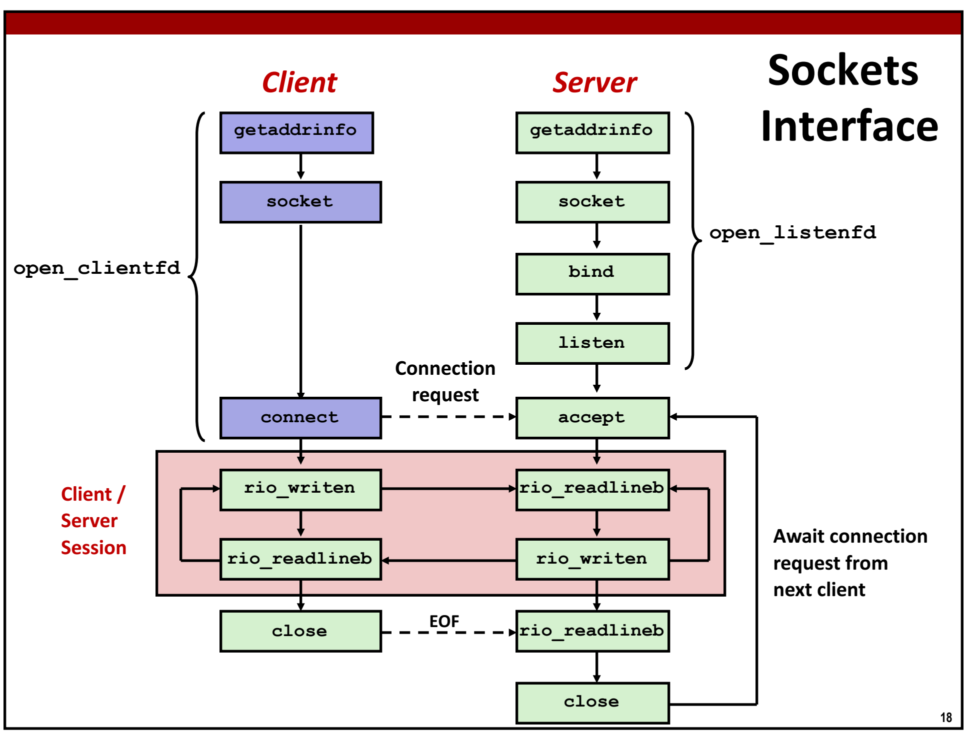
2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client
- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request


Sockets Interface
Client
Server
getaddrinfo
socket
connect
open_clientfd
getaddrinfo
socket
bind
listen
accept
open_listenfd
Connection request
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client
17


Sockets Interface
Client
Server
getaddrinfo
socket
connect
open_clientfd
getaddrinfo
socket
bind
listen
accept
open_listenfd
Connection request
Client / Server Session
rio_writen
rio_readlineb
rio_readlineb
rio_writen
close
EOF
rio_readlineb
close
Await connection request from next client
18

Host and Service Conversion: getaddrinfo

- **getaddrinfo is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.**
  - Replaces obsolete gethostbyname and getservbyname funcs.

- **Advantages:**
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

- **Disadvantages**
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

19

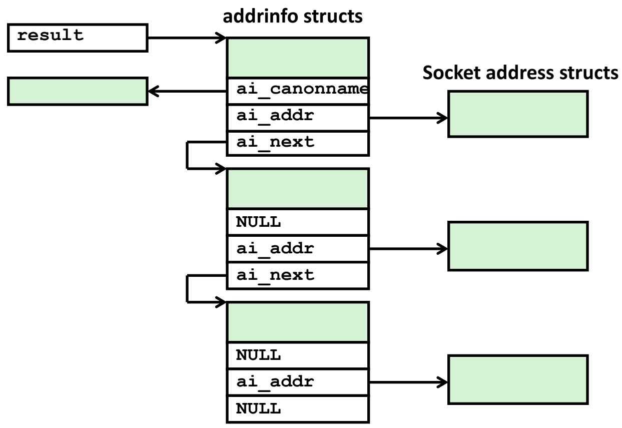
Host and Service Conversion: getaddrinfo

```
int getaddrinfo(const char *host,            /* Hostname or address */
                const char *service,         /* Port or service name */
                const struct addrinfo *hints,/* Input parameters */
                struct addrinfo **result);   /* Output linked list */

void freeaddrinfo(struct addrinfo *result);  /* Free linked list */

const char *gai_strerror(int errcode);       /* Return error msg */
```

- **Given host and service, getaddrinfo returns result that points to a linked list of addrinfo structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.**
- **Helper functions:**
  - freeadderinfo frees the entire linked list.
  - gai_strerror converts error code to an error message.

20

Linked List Returned by `getaddrinfo`



- **Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.**
- **Servers: walk the list until calls to `socket` and `bind` succeed.**

`addrinfo` Struct

```
struct addrinfo {
    int              ai_flags;     /* Hints argument flags */
    int              ai_family;    /* First arg to socket function */
    int              ai_socktype;  /* Second arg to socket function */
    int              ai_protocol;  /* Third arg to socket function  */
    char            *ai_canonname; /* Canonical host name */
    size_t           ai_addrlen;   /* Size of ai_addr struct */
    struct sockaddr *ai_addr;      /* Ptr to socket address structure */
    struct addrinfo *ai_next;      /* Ptr to next item in linked list */
};
```

- **Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to `socket` function.**
- **Also points to a socket address struct that can be passed directly to `connect` and `bind` functions .**

Host and Service Conversion: getnameinfo

- **getnameinfo is the inverse of getaddrinfo, converting a socket address to the corresponding host and service.**
  - Replaces obsolete gethostbyaddr and getservbyport funcs.
  - Reentrant and protocol independent.

```
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
                char *host, size_t hostlen,    /* Out: host */
                char *serv, size_t servlen,    /* Out: service */
                int flags);                    /* optional flags */
```

23

Conversion Example

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
```

hostinfo.c

24

Conversion Example (cont)

```
    /* Walk the list and display each IP address */
    flags = NI_NUMERICHOST; /* Display address instead of name */
    for (p = listp; p; p = p->ai_next) {
        Getnameinfo(p->ai_addr, p->ai_addrlen,
                    buf, MAXLINE, NULL, 0, flags);
        printf("%s\n", buf);
    }

    /* Clean up */
    Freeaddrinfo(listp);

    exit(0);
}
```

hostinfo.c

Running hostinfo

```
whaleshark> ./hostinfo localhost
127.0.0.1

whaleshark> ./hostinfo www.cs.gmu.edu
129.174.125.153

whaleshark> ./hostinfo twitter.com
199.16.156.230
199.16.156.38
199.16.156.102
199.16.156.198
```

Sockets Helper: open_clientfd

- Establish a connection with a server

```
int open_clientfd(char *hostname, char *port) {
  int clientfd;
  struct addrinfo hints, *listp, *p;

  /* Get a list of potential server addresses */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
  hints.ai_flags = AI_NUMERICSERV;  /* …using numeric port arg. */
  hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
  Getaddrinfo(hostname, port, &hints, &listp);
```

csapp.c

27

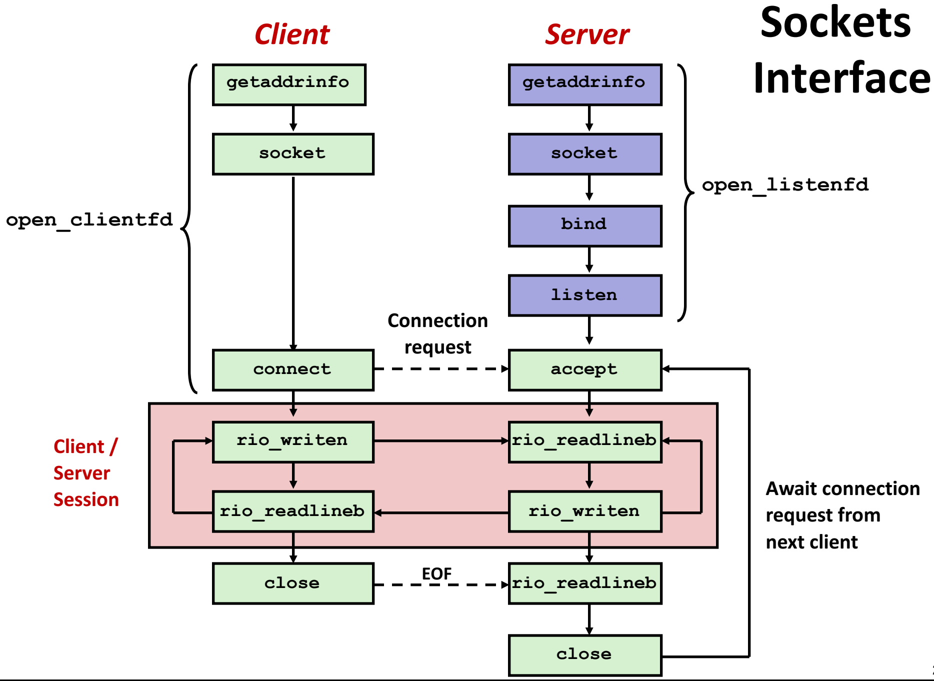
Sockets Helper: open_clientfd (cont)

```
    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        Close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```

csapp.c

28

Sockets Interface



Sockets Helper: `open_listenfd`

- Create a listening descriptor that can be used to accept connection requests from clients.

```
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;              /* Accept connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* …on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;            /* …using port no. */
    Getaddrinfo(NULL, port, &hints, &listp);
```

csapp.c

Sockets Helper: open_listenfd (cont)

```
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype,
                               p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }
```

csapp.c

Sockets Helper: open_listenfd (cont)

```
    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept conn. requests */
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd);
        return -1;
    }
    return listenfd;
}
```

csapp.c

- **Key point:** open_clientfd and open_listenfd are both independent of any particular version of IP.

Echo Client: Main Routine

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

echoclient.c

Using read() and write() for network communication

```
#define BLEN 120
char *req = ''request of some sort'';
char buf[BLEN];
char *bptr;
int n;
int buflen;

bptr = buf;
buflen = BLEN;

/* send request */

write(s,req,strlen(req);

/* read response (may come in several pieces) */

while ((n = read(s,bptr,buflen) > 0) {
   bptr += n;
   buflen -= n;
}
```

Need to handle short counts!!

The RIO Package

- RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts

- RIO provides two different kinds of functions
  - Unbuffered input and output of binary data
    - `rio_readn` and `rio_writen`
  - Buffered input of text lines and binary data
    - `rio_readlineb` and `rio_readnb`
    - Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor

- Download from http://csapp.cs.cmu.edu/3e/code.html
  → `src/csapp.c` and `include/csapp.h`

Iterative Echo Server: Main Routine

```
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echoserveri.c

Echo Server: echo function

- **The server uses RIO to read and echo text lines until EOF (end-of-file) condition is encountered.**
  - EOF condition caused by client calling **close(clientfd)**

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```
echo.c

Testing Servers Using telnet

- **The telnet program is invaluable for testing servers that transmit ASCII strings over Internet connections**
  - Our simple echo server
  - Web servers
  - Mail servers

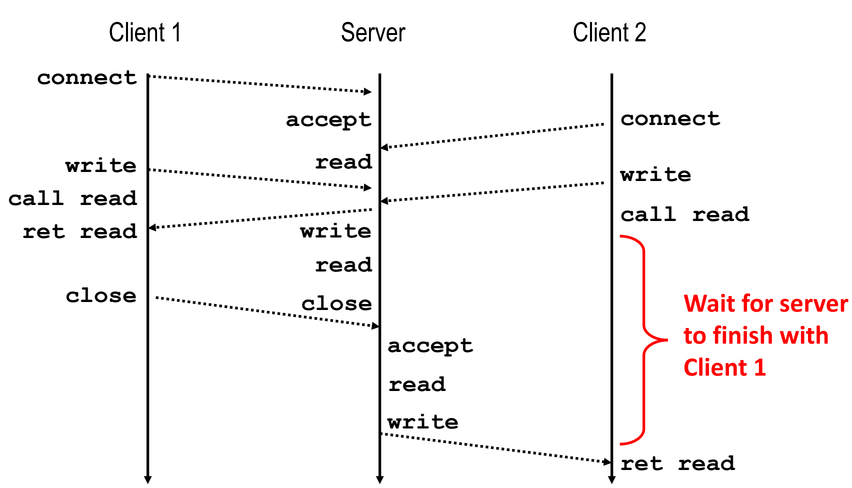
- **Usage:**
  - **linux> *telnet <host> <portnumber>***
  - Creates a connection with a server running on ***<host>*** and listening on port ***<portnumber>***

Testing the Echo Server With `telnet`

```
whaleshark> ./echoserveri 15213
Connected to (MAKOSHARK.ICS.CS.CMU.EDU, 50280)
server received 11 bytes
server received 8 bytes



makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
Hi there!
Hi there!
Howdy!
Howdy!
^]
telnet> quit
Connection closed.
makoshark>
```

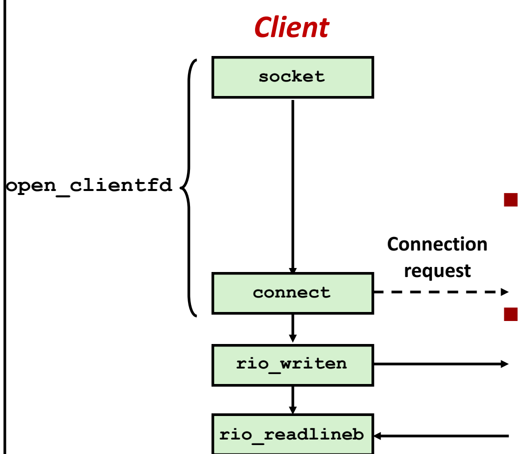

Iterative Servers

- Iterative servers process one request at a time



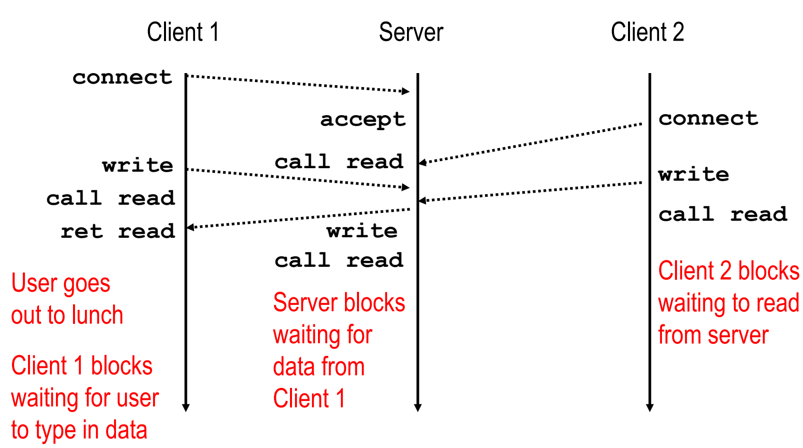
Where Does Second Client Block?

- **Second client attempts to connect to iterative server**



- **Call to connect returns**
  - Even though connection not yet accepted
  - Server side TCP manager queues request
  - Feature known as "TCP listen backlog"
- **Call to rio_writen returns**
  - Server side TCP manager buffers input data
- **Call to rio_readlineb blocks**
  - Server hasn't written anything for it to read yet.

Fundamental Flaw of Iterative Servers



- **Solution: use *concurrent servers* instead**
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

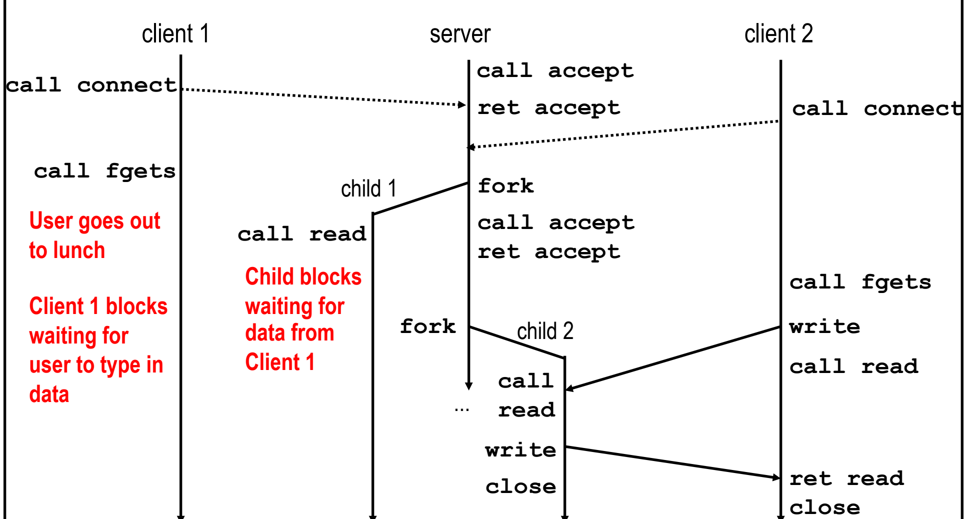
1. Process-based
- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based
- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Approach #1: Process-based Servers

- **Spawn separate process for each client**



Process-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

45

Process-Based Concurrent Echo Server (cont)

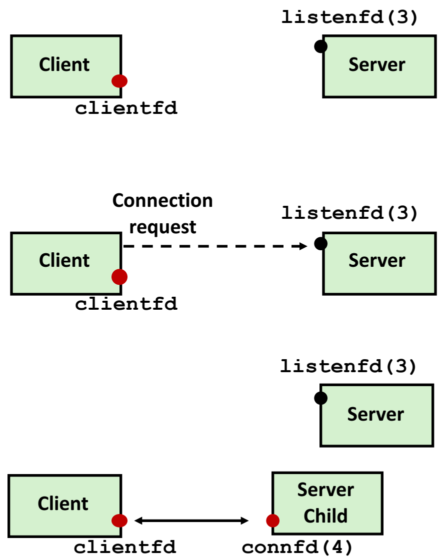
```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

echoserverp.c

- Reap all zombie children

46

Concurrent Server: `accept` Illustrated



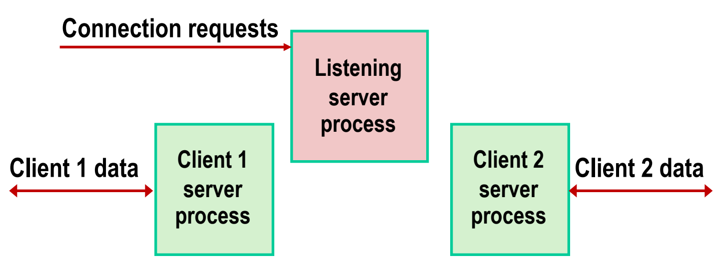
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling `connect`

3. Server returns `connfd` from `accept`. Forks child to handle client. Connection is now established between `clientfd` and `connfd`

47

Process-based Server Execution Model



- Each client handled by independent child process
- No shared state between them
- Both parent & child have copies of listenfd and connfd
  - Parent must close `connfd`
  - Child should close `listenfd`

48

Issues with Process-based Servers

- **Listening server process must reap zombie children**
  - to avoid fatal memory leak
- **Parent process must `close` its copy of `connfd`**
  - Kernel keeps reference count for each socket/open file
  - After fork, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) = 0`

49

Pros and Cons of Process-based Servers

- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - Requires IPC (interprocess communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

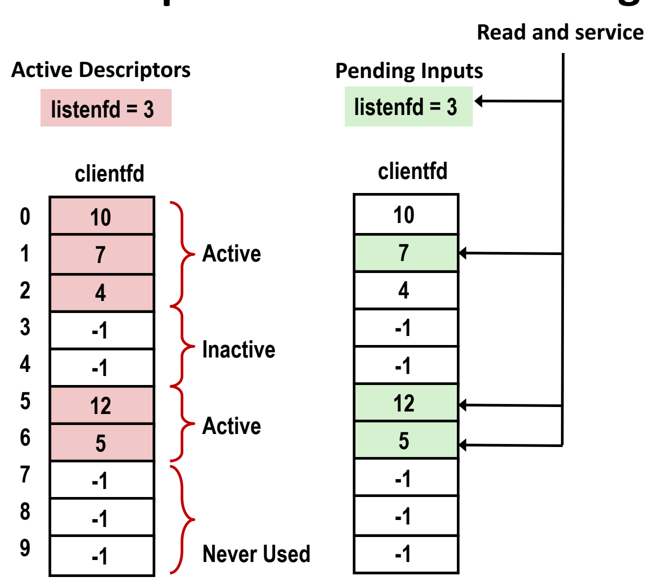
50

Approach #2: Event-based Servers

- **Server maintains set of active connections**
  - Array of connfd's
- **Repeat:**
  - Determine which descriptors (connfd's or listenfd) have pending inputs
    - e.g., using `select` or `epoll` functions
    - arrival of pending input is an *event*
  - If listenfd has input, then `accept` connection
    - and add new connfd to array
  - Service all connfd's with pending inputs
- **Details for select-based server in book**

51

I/O Multiplexed Event Processing



52

Pros and Cons of Event-based Servers

- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- **– Significantly more complex to code than process- or thread-based designs.**
- **– Hard to provide fine-grained concurrency**
  - E.g., how to deal with partial HTTP request headers
- **– Cannot take advantage of multi-core**
  - Single thread of control

53

Approach #3: Thread-based Servers

- **Very similar to approach #1 (process-based)**
  - …but using threads instead of processes

54

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                   (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```
echoservert.c

- **`malloc` of connected descriptor necessary to avoid deadly race (later)**

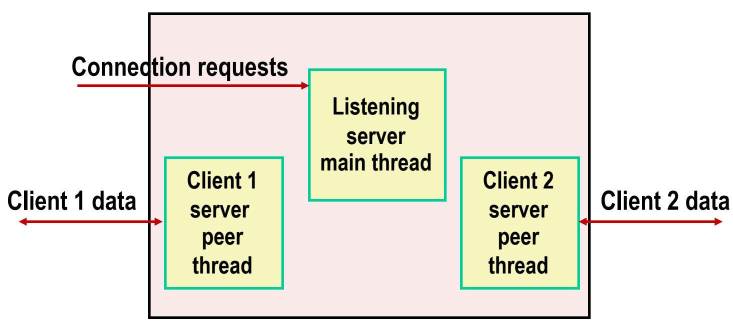
Thread-Based Concurrent Server (cont)

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```
echoservert.c

- Run thread in "detached" mode.
  - Runs independently of other threads
  - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold `connfd`.
- Close `connfd` (important!)

57

Thread-based Server Execution Model



- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

58

Issues With Thread-Based Servers

- **Must run "detached" to avoid memory leak**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable
    - use `pthread_detach(pthread_self())` to make detached
- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread's stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
- **All functions called by a thread must be *thread-safe***

59

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**
- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!

60

Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- **Event-based (discussed in book)**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core
- **Thread-based**
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable