CS 344: Operating Systems I

02.15: Part III: Sockets

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Notice

- Announcements
  - Sanghyun's office hours will be on the 16th at 11:00 am to 12:30 pm
    - No office hours on the 17th

Topics we left

- Part III: IPC, RPC, and Networking
  - Motivation
    - What is IPC/RPC?
    - Why do we need IPC/RPC?
  - Provide abstractions
    - What is the mechanisms OS support for IPC?
  - Offer standard interface
    - How can we use a signal?
    - How can we use a pipe?
  - Manage resources
    - (Overview) How does OS support these mechanisms?

Manage resources: signal internals

- Signal from Process A -> Process B
  - **OS kernel**
    - Checks if Process B has pending signals
    - Pauses the execution of Process B
    - Invokes do_signal()
    - do_signal() call invokes handle_signal()
  - **Process B**
    - Run code in signal_handler
    - Return back to kernel: sigreturn()
  - **OS Kernel**
    - Resume Process B

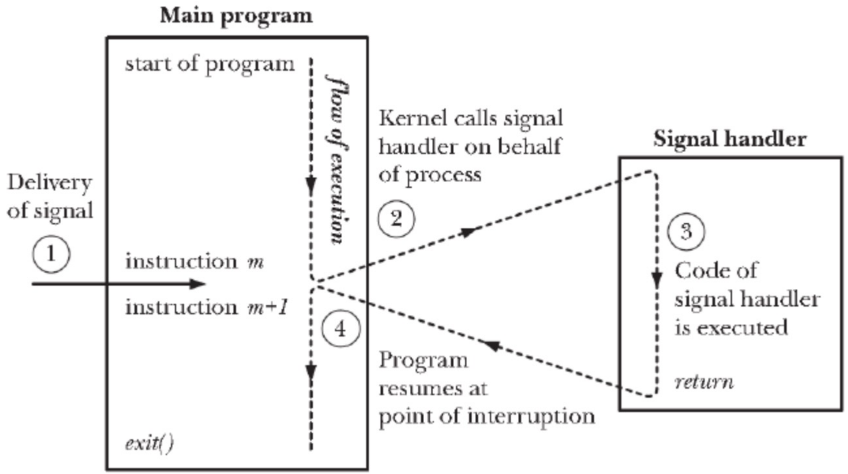


Figure 20-1: Signal delivery and handler execution

Manage resources: signal managed by OS

- Mechanism (OS-level)
  - Process A sends a signal to Process B
  - OS kernel updates B's process context **(Send)**
  - OS kernel asks B to react to the signal **(Receive)**
    - Process B will execute a signal handler
    - Process B declines to receive the signal
  - Multiple processes send signals to B **(Pending)**
    - Up to 1 pending signal per type for each process
    - More signals of the same type will be discarded

```
	/* Signal handlers: */
	struct signal_struct		*signal;
	struct sighand_struct __rcu		*sighand;
	sigset_t			blocked;
	sigset_t			real_blocked;
	/* Restored if set_restore_sigmask() was used: */
	sigset_t			saved_sigmask;
	struct sigpending		pending;
	unsigned long			sas_ss_sp;
	size_t				sas_ss_size;
	unsigned int			sas_ss_flags;

	struct callback_head		*task_works;

#ifdef CONFIG_AUDIT
#ifdef CONFIG_AUDITSYSCALL
	struct audit_context		*audit_context;
#endif
	kuid_t				loginuid;
	unsigned int			sessionid;
#endif
	struct seccomp			seccomp;
	struct syscall_user_dispatch	syscall_dispatch;
```

Manage resources: pipe

- Data structure
  - **Queue** in memory
  - **(Rule)** If Proc A writes data, the data will be in the kernel queue until Proc B reads it

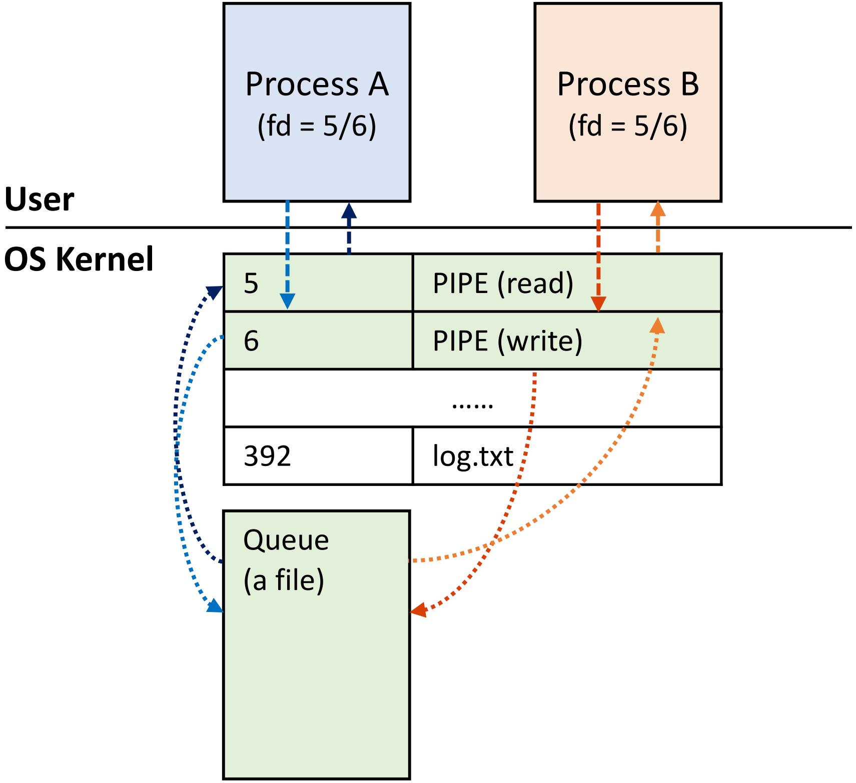
- OS kernel's queue control:
  - Queue can be **full/empty**
    - If the queue is full, OS kernel asks Proc A (write) to wait
    - If the queue is empty, OS kernel asks Proc B (read) to wait

Topics for today

- Part III: Sockets
  - Motivation
    - Why do we need RPC?
  - Provide abstraction
    - What is the mechanism OS support for RPC?
  - Offer standard interface
    - How can we use a socket(s)?
  - Manage resources
    - (Not in this lecture) How does OS support the socket?

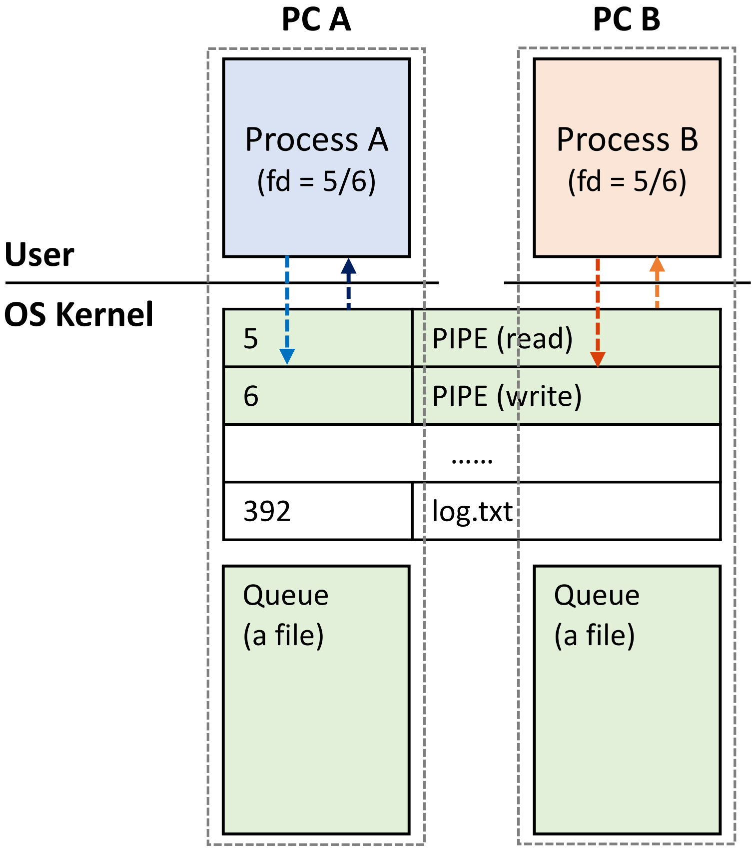
Motivation: RPC

- Pipe only support IPCs
  - What if Proc A and B are running on different hosts (or machines)?



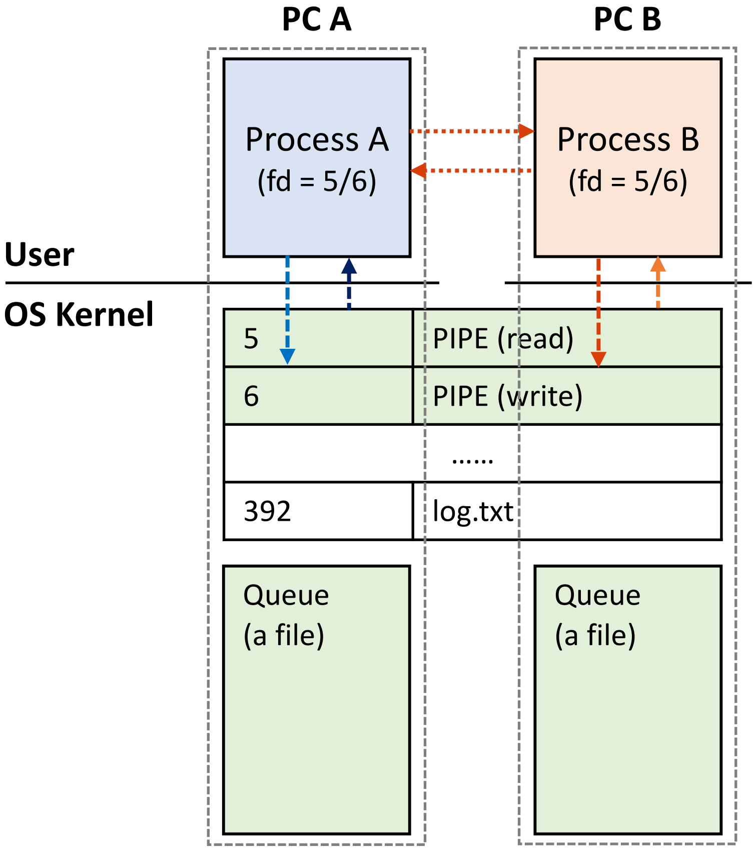
Motivation: RPC design

- Pipe only support IPCs
  - What if Proc A and B are running on different hosts (or machines)?
- **Solution approach**
  - Each process has its own queue



Motivation: RPC design

- Pipe only support IPCs
  - What if Proc A and B are running on different hosts (or machines)?
- **Solution approach**
  - Each process has its own queue
  - Design a communication *protocol*(s)



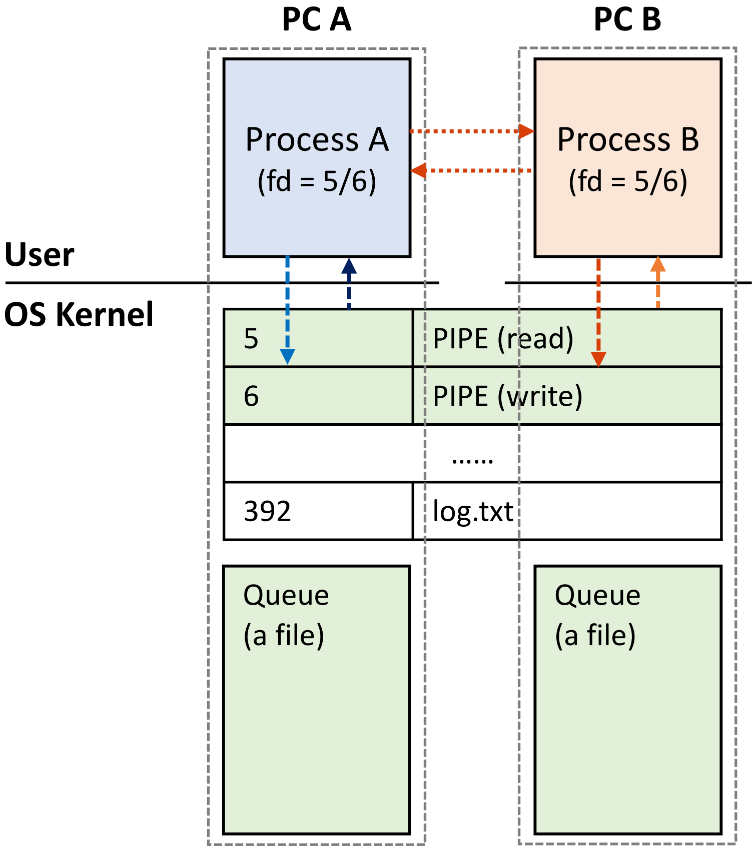
Motivation: RPC protocol design

- **Caller (You)**
  - Open up your phone
  - Search a restaurant's phone number
  - Call and wait
  - I'd like to have a table for two today at 7 pm
  - John Doe
  - 123-456-7890
  - Thank you
  - Hang up
- **Callee (XYZ Restaurant)**
  - Thank you for calling XYZ. How can I help you?
  - Two at 7 pm. Yes, we have a table.
  - May I have the name on the reservation?
  - and a phone number?
  - Today, 7 pm today, John Doe. You're all set.
  - Thank you. See you soon.
  - Bye

MOTIVATION: RPC PROTOCOL DESIGN

- Pipe only support IPCs
  - What if Proc A and B are running on different hosts (or machines)?
  - What if there are multiple hosts?

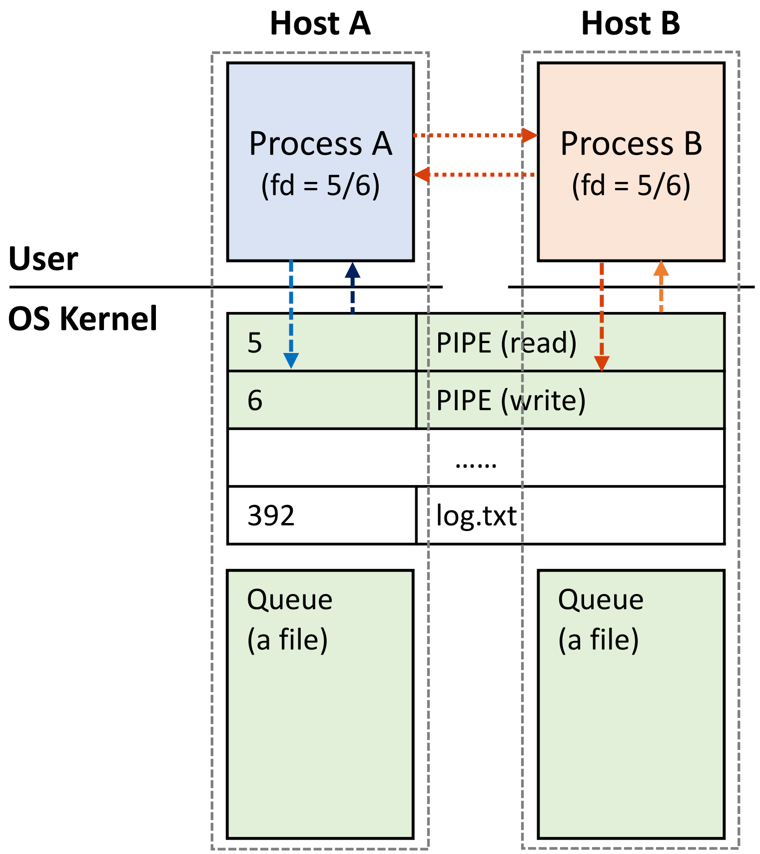
- **Solution approach**
  - Each process has its own queue
  - Design a communication *protocol*(s)



Motivation: RPC protocol design

- Pipe only support IPCs
  - What if Proc A and B are running on different hosts (or machines)?
  - What if there are multiple hosts?

- **Solution approach**
  - Each process has its own queue
  - Design a communication *protocol*(s)
  - Require an *address* for each host (like a phone number for the restaurant)



Topics for today

- Part III: Sockets
  - Motivation
    - Why do we need RPC?
  - Provide abstraction
    - What is the mechanism OS support for RPC?
  - Offer standard interface
    - How can we use a socket(s)?
  - Manage resources
    - (Not in this lecture) How does OS support the socket?

Provide abstraction: socket

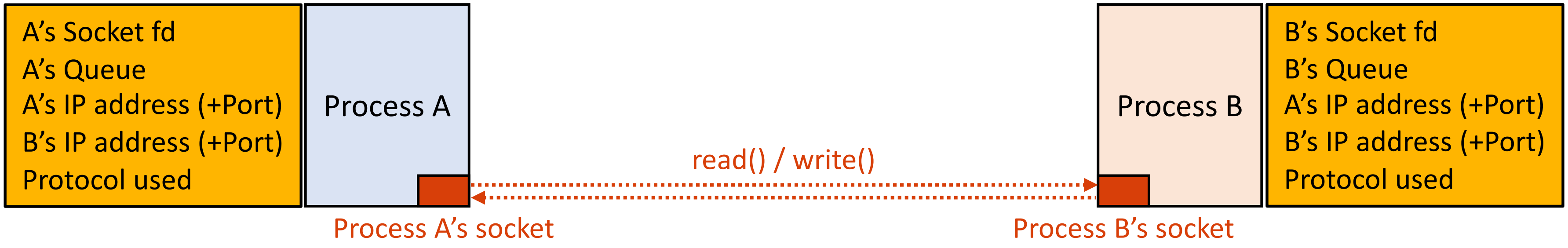
- Socket
  - **Definition:** an *abstract* structure for sending and receiving data
  - **TL; DR:** a *bi-directional* pipe

- Socket components
  - A structure (① a file descriptor and ② a queue)
  - IP addresses (③ source and ④ destination addresses)
  - ⑤ Protocols (*e.g.*, TCP/IP or UDP) to use

Provide abstraction: socket

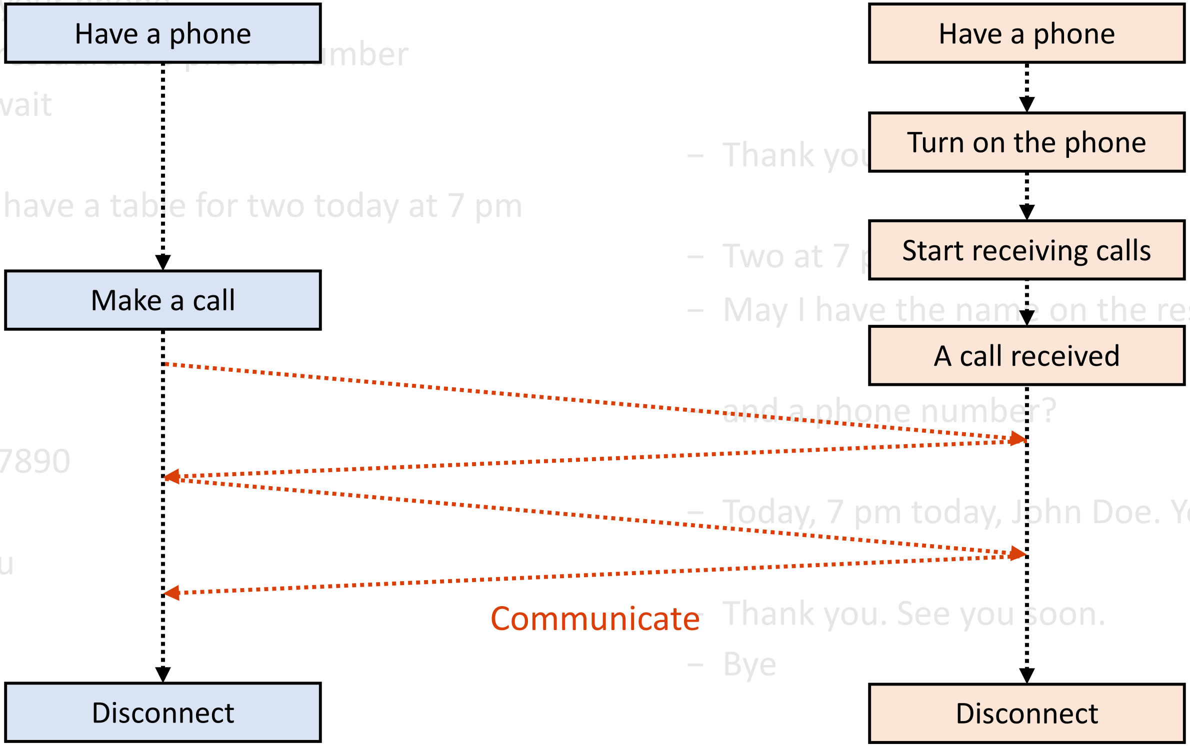
- Socket
  - **Definition:** an *abstract* structure for sending and receiving data
  - **TL; DR:** a *bi-directional* pipe

- Socket components
  - A structure (① a file descriptor and ② a queue)
  - IP addresses (③ source and ④ destination addresses)
  - ⑤ Protocols (*e.g.*, TCP/IP or UDP) to use



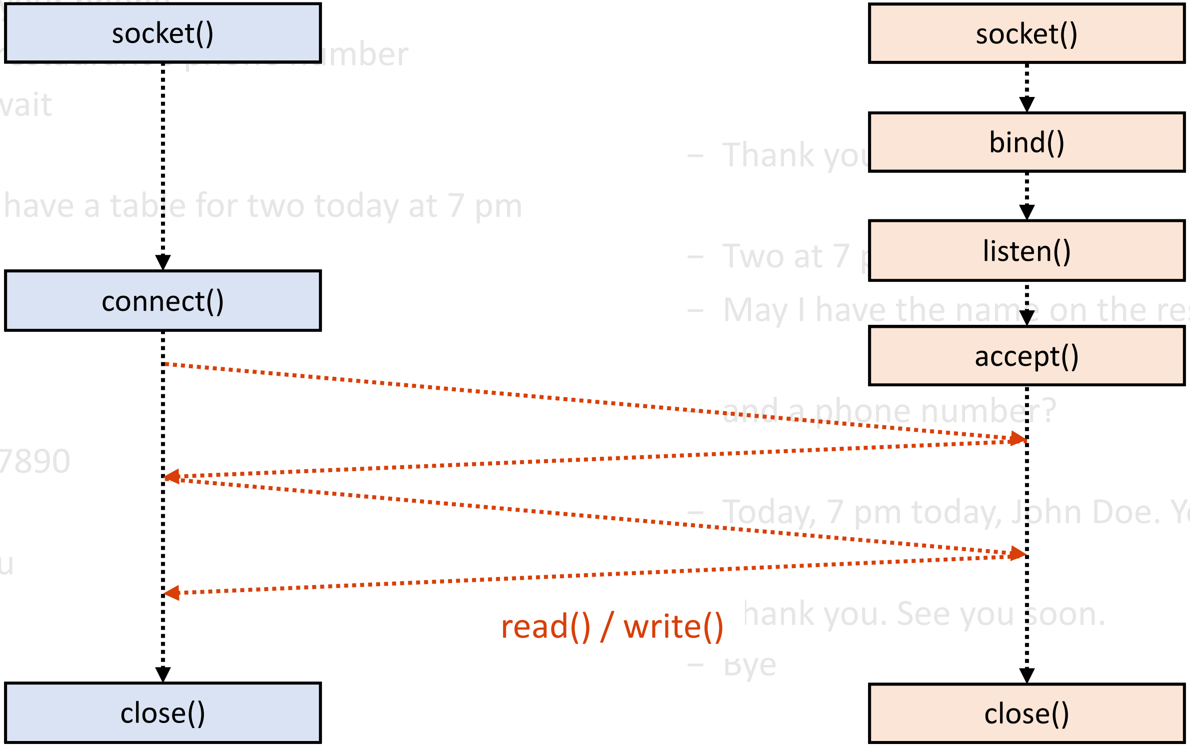
Provide abstraction: socket programming

- **Caller (You: client)**
- **Callee (XYZ Restaurant: server)**



Provide abstraction: socket programming

- **Caller (You: client)**
  - Open up [illegible]
  - Search a [illegible] mber
  - Call and wait
  - I'd like to have a table for two today at 7 pm
  - John Doe
  - 123-456-7890
  - Thank you
  - Hang up
- **Callee (XYZ Restaurant: server)**
  - Thank you [illegible] can I help you?
  - Two at 7 [illegible]
  - May I have the name on the reservation?
  - and a phone number?
  - Today, 7 pm today, John Doe. You're all set.
  - [illegible] hank you. See you soon.
  - Bye



Topics for today

- Part III: Sockets
  - Motivation
    - Why do we need RPC?
  - Provide abstraction
    - What is the mechanism OS support for RPC?
  - Offer standard interface
    - How can we use a socket(s)?
  - Manage resources
    - (Not in this lecture) How does OS support the socket?

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Domain	Descriptions
AF_UNIX	local communication
AF_LOCAL	synonym for AF_UNIX
AF_INET	IPv4 Internet protocol
AF_INET6	IPv6 Internet protocol
AF_PACKET	low-level communication protocol
…	

Type	Descriptions
SOCK_STREM	byte streams
SOCK_RAW	raw network protocol access
…	

Protocol	typically 0

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Level	**Descriptions**
SOL_SOCKET	to set the socket option
IPPROTO_TCP	to interpret the option as TCP
…	

Option Name	**Descriptions**
SO_DEBUG	turn on recording of debug info
SO_BROADCAST	broadcast messages (e.g., UDP)
SO_KEEPALIVE	keeps connection alive
…	

OPTVAL
OPTLEN Please refer to this man page (link)

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Argument	Descriptions
addr	IPv4/v6 address structure
addrlen	"sizeof" the above structure
…	

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Argument	**Descriptions**
backlog	max number of waiting connections
…	

Offer standard interface: socket

- Socket system calls
  - int socket(int domain, int type, int protocol);
  - int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - int listen(int sockfd, int backlog);
  - int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

Argument	Descriptions
addr	IPv4/v6 address structure (client)
addrlen	“sizeof” the above structure (client)
…	

Offer standard interface: server.c

```
... omit the includes

#define  BUF_SIZE          1024
#define  PORT              8080

int main(void) {
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   char buffer[BUF_SIZE] = { 0 };
   char* hello = "Hello (server)!";

   // create socket (returns a sockfd for reading/writing)
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
      perror("socket failed");
      exit(EXIT_FAILURE);
   }

   // configure the socket by setting the options
   if (setsockopt(server_fd, SOL_SOCKET,
                        SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
      perror("setsocketopt failed");
      exit(EXIT_FAILURE);
   }

   address.sin_family = AF_INET;
   address.sin_addr.s_addr = INADDR_ANY;  // bind to any address
   address.sin_port = htons(PORT);          // format the port num

   // attach socket to the port 8080
   if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
      perror("bind failed");
      exit(EXIT_FAILURE);
   }

   if (listen(server_fd, 3) < 0) {
      perror("listen failed");
      exit(EXIT_FAILURE);
   }

   if ((new_socket = accept(server_fd,
                           (struct sockaddr*)&address,
                           (socklen_t*)&sizeof(address))) < 0) {
      perror("accept");
      exit(EXIT_FAILURE);
   }

   valread = read(new_socket, buffer, 1024);
   printf("%s\n", buffer);
   send(new_socket, hello, strlen(hello), 0);
   printf("Message sent (server)\n");
   return 0;
}
```

AF_INET (IPv4)
SOCK_STREAM (bi-directional)

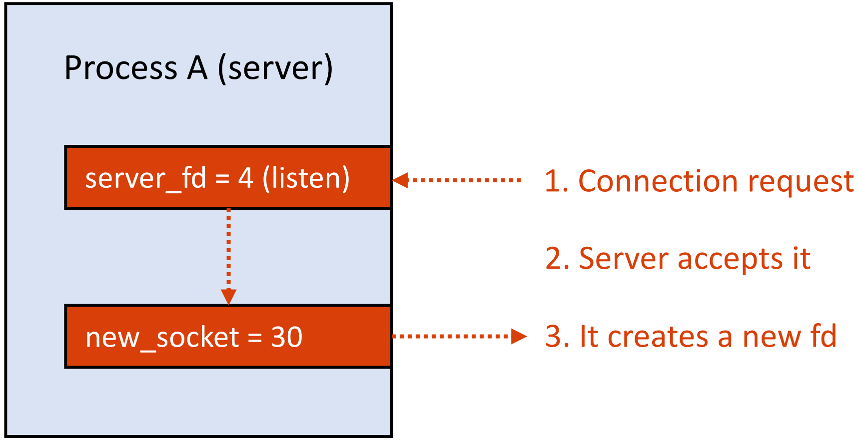
SO_REUSEADDR
SO_REUSEPORT
opt (optional value)

Bind the socket to the address
> Any IP (of the host)
> Port # 8080

Listen incoming connections
> Use the socket fd
> Allow 3 connections (max.)

Start accepting connections
> Use the socket fd
> Use the address specified
> Return the fd (accepted)

Offer standard interface: server.c



socket fd != new_socket

Design choice:
We want to *separate* the file descriptor for listening connection requests (socket_fd) from the file descriptor used for communicating with the client (new_socket)

Bind the socket to the address
> Any IP (of the host)
> Port # 8080

```c
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;  // bind to any address
address.sin_port = htons(PORT);        // format the port num

// attach socket to the port 8080
if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
   perror("bind failed");
   exit(EXIT_FAILURE);
}

if (listen(server_fd, 3) < 0) {
   perror("listen failed");
   exit(EXIT_FAILURE);
}
```

Listen incoming connections
> Use the socket fd
> Allow 3 connections (max.)

```c
if ((new_socket = accept(server_fd,
                         (struct sockaddr*)&address,
                         (socklen_t*)&sizeof(address))) < 0) {
   perror("accept");
   exit(EXIT_FAILURE);
}

valread = read(new_socket, buffer, 1024);
printf("%s\n", buffer);
send(new_socket, hello, strlen(hello), 0);
printf("Message sent (server)\n");
return 0;
}
```

Start accepting connections
> Use the socket fd
> Use the address specified
> Return the fd (accepted)

Offer standard interface: client.c

```c
#define  IPADDR  "127.0.0.1"
#define  PORT     8080
#define  BUFSIZE 1024

int main(void)
{
   int sock = 0, valread;
   struct sockaddr_in serv_addr;
   char* hello = "Hello (client)";
   char buffer[BUFSIZE] = { 0 };

   // create a socket
   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      printf("Error: socket creation error\n");
      return -1;
   }

   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(PORT);

   // convert IP addresses from text to binary
   if (inet_pton(AF_INET, IPADDR, &serv_addr.sin_addr) <= 0) {
      printf("Error: invalid address, address not supported\n");
      return -1;
   }

   if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
      printf("Connection Failed\n");
      return -1;
   }

   send(sock, hello, strlen(hello), 0);
   printf("Message sent (client)\n");
   valread = read(sock, buffer, BUFSIZE);
   printf("%s\n", buffer);

   return 0;
}
```

AF_INET (IPv4)
SOCK_STREAM (bi-directional)

Connect to the server, running on the IP address we specify “127.0.0.1”

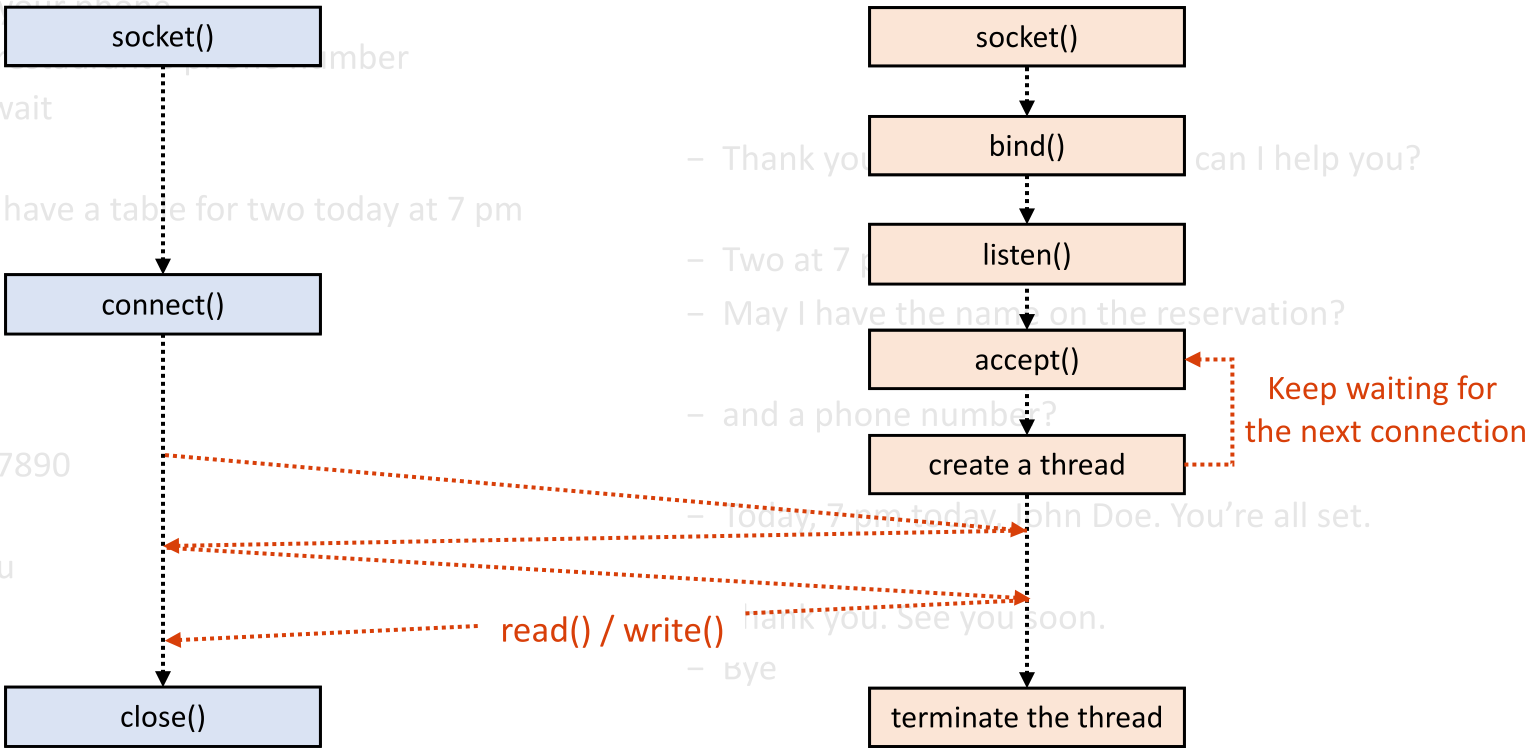
Execution result

```
$ gcc -o server server.c
$ gcc -o client client.c
$ ./server &
$ ./client

Message sent (client)
Hello (client)
Message sent (server)
Hello (server)
```

Offer standard interface: socket w. multiple connections

- **Caller (Client)**
- **Callee (Server)**



OFFER STANDARD INTERFACE: MULTI_THREADED_SERVER.C

… omit the code for creating and binding sockets …

```
if (listen(server_fd, 100) < 0) {
    perror("listen failed");
    exit(EXIT_FAILURE);
}

struct sockaddr_in client;
int c = sizeof(struct sockaddr_in);
pthread_t conn_threads[100];

while (1) {
    if (csocket = accept(server_fd,
                         (struct sockaddr *)&client,
                         (socklen_t *)&c)) < 0) {
        printf("Server waits for a connection\n");
        continue;
    }

    printf("Server accepts the connection\n");
    if (pthread_create(&tid, NULL, conn_handler, (void *)&csocket)  < 0) {
        perror("Error: cannot start a thread");
        exit(EXIT_FAILURE);
    }
}

return 0;        // this thread infinitely runs, this line won't be reached
```

Listen; up to 100 connections

Keep waiting

Once csocket is greater than 0, then it proceeds to the next line

Create a threat that handles the communications between this server and the connected client (we pass socket_desc as an arg)

```
void *conn_handler(void *socket_desc) {
    int rlen;
    char buffer[BUFSIZE] = { 0 };
    char *ack = "Received";

    while (rlen = read(socket_desc, buffer, BUFSIZE)) {
        if (rlen < 0) continue;

        printf("(server) Received: %s\n", buffer);

        send(socket_desc, ack, strlen(ack), 0);
        printf("(server) Ack sent\n");
    }

}
```

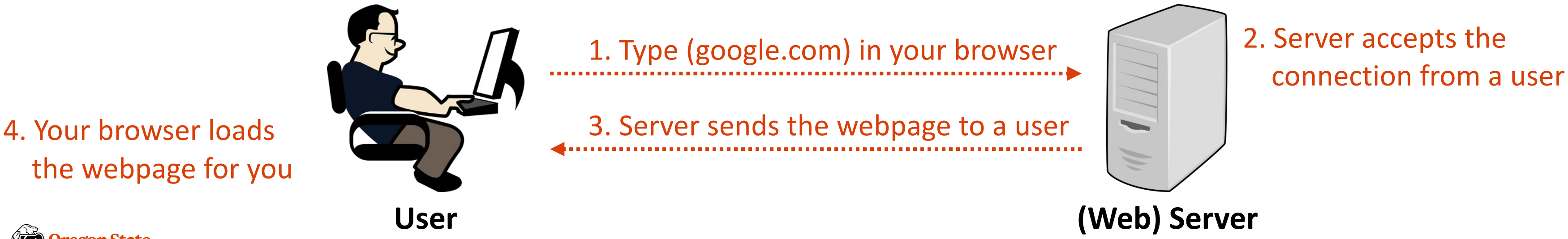
Keep reading data from the socket
1. If no data, it continues
2. If data is, it prints out the data
3. Then it sends the ack message

Socket programming example

- Linux daemons
  - **Linux daemon:** a Linux process runs in the background
  - How it *mostly* works:
    - Daemons start when we boot an OS and wait for our connections
    - We connect to daemons and use their functionalities
  - Example daemons:
    - httpd: web server daemon
    - ftpd: FTP server daemon
    - mysql: MySQL database server daemon
    - sshd: secure shell daemon
    - … (you can find them; most daemons end with “d”)

Socket programming example

- Client-server architecture
  - **An example HTTP server**
  - How HTTP server works:
    - A server spins up and waits for connections
    - A user requests the webpages to the server
    - A server accepts this connection and sends HTTP webpages to the user
    - (Mostly) The webpages contain code for interactions (*e.g.*, JavaScripts)
      - A user clicks a butten (or advertisements), the browser sends a request to the server
      - The browser does appropriate actions and sends a new webpage containing the results



Topics for today

- Part III: Sockets
  - Motivation
    - Why do we need RPC?
  - Provide abstraction
    - What is the mechanism OS support for RPC?
  - Offer standard interface
    - How can we use a socket(s)?
  - Manage resources
    - (Not in this lecture) How does OS support the socket?

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab