Interprocess Communication (IPC)

Methods

- Shared memory
- Shared files
- Signals
- Message queues
- Pipes
  - Ordinary pipes
  - Named pipes
- Sockets

Shared Memory

Shared Memory

- Unix kernel prohibits one process from accessing (reading, writing) memory belonging to another process
  - System V shared memory allows a process to read and/or write to memory created by another
  - POSIX shared memory alternative is more portable
- Advantages:
  - Random access: you can update a small piece in the middle of a data structure, rather than the entire structure
  - Efficiency: directly accessed as it resides in the user process memory
- Disadvantages:
  - No automatic synchronization as in pipes or message queues
  - Pointers are only valid within a given process. Thus, pointer offsets cannot be assumed to be valid across inter-process boundaries. This complicates the sharing of linked lists or binary trees.

Example

```c
#define SHM_NAME "my-shm"

int sid = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);

int size = 2 * sizeof(int);
ftruncate(sid, size);

if (fork() == 0) {  // child process
    int* p = (int*)mmap(NULL, size, PROT_WRITE, MAP_SHARED, sid, 0);
    p[0] = 10;
    p[1] = 20;

} else {    // parent process
    wait(NULL);

    int* p = (int*)mmap(NULL, size, PROT_READ, MAP_SHARED, sid, 0);
    int x = p[0];
    int y = p[1];
    printf("x + y = %d\n", x + y);

    shm_unlink(SHM_NAME);
}
```

Compile with `-lrt` flag:

```
?> gcc shared-mem.c -o shared-mem -lrt
```

Shared Files

Shared Files

- Files can be accessed by multiple processes
- Advantages:
  - Simple
  - Portable
- Disadvantages:
  - Files resident in external memory are generally
  - Data may be read by external users/programs if no measure for control of permission is taken

Example

```c
#define FILENAME "/tmp/shared-file"

if (fork() == 0) {  // child process
    int x = 10, y = 20;

    int fd = open(FILENAME, O_CREAT | O_WRONLY);
    write(fd, &x, sizeof(x));
    write(fd, &y, sizeof(y));
    close(fd);

} else {    // parent process
    wait(NULL);

    int x, y;
    int fd = open(FILENAME, O_RDONLY);
    int n1 = read(fd, &x, sizeof(x));
    int n2 = read(fd, &y, sizeof(y));
    close(fd);

    printf("x + y = %d\n", x + y);
}
```

Signals

Signals

- A signal is a notification to a process indicating the occurrence of an event
- Signal is also called software interrupt and is not predictable to know its occurrence, hence it is also called an asynchronous event
- Signal can be specified with a number or a name
  - Some are well-defined: `SIGKILL` (9), `SIGALRM` (14), `SIGTERM` (15),…
  - Some are left for user to define: `SIGUSR1` (10), `SIGUSR2` (12)
- When a signal is sent
  - The OS interrupts the target process' normal flow of execution to deliver the signal
  - If the process has previously registered a signal handler, that routine is executed
  - `SIGKILL` and `SIGSTOP` cannot be caught or ignored

Usage

- Set the handler for a signal:
  - `signal_handler_t signal(int signum, signal_handler_t func)`
  - `func` can be:
    - A user-defined handler function
    - `SIG_IGN`: Ignore the signal
    - `SIG_DFL`: Use the default handler
  - Returns the previous signal handler function
- Send a signal:
  - `int kill(pid_t pid, int signum)`
- Suspend a process until a signal arrives:
  - `int pause()`

Examples

- Avoiding Ctrl+C and Ctrl+Z:
  - ```
    signal(SIGINT, SIG_IGN);
    signal(SIGTSTP, SIG_IGN);
    ```
- Synchronization of processes:
  - ```
    void sig_handler_parent(int signum) {
        printf("Parent: Received a response signal from child\n");
    }

    void sig_handler_child(int signum) {
        printf("Child: Received a signal from parent\n");
        kill(getppid(), SIGUSR1);
    }

    if (fork() == 0) {   // child process
        signal(SIGUSR1, sig_handler_child);
        pause();
    } else {    // parent process
        signal(SIGUSR1, sig_handler_parent);
        sleep(1);
        kill(pid, SIGUSR1);
        pause();
    }
    ```

Message Queues

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- New messages are added to the end of a queue
  - `int msgsnd(int msgid, const void* msg, size_t size, int flags)`
  - A message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length)
- Messages can then be fetched by another process
  - `ssize_t msgrcv(int msgid, void* msg, size_t size, long type, int flags)`
  - Pass `0` as `type` to fetch the 1st message of any type, or a positive value for the desired message type

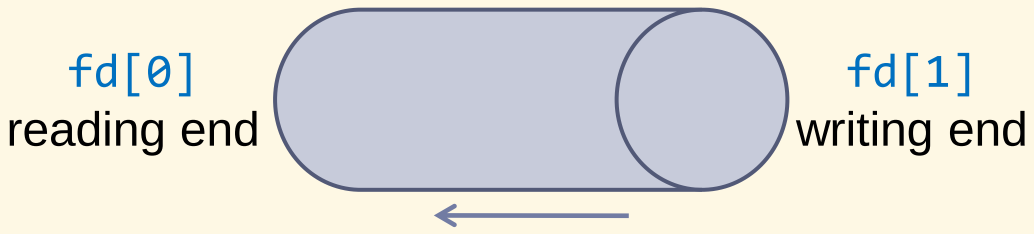
Example

```c
#define MQ_PATHNAME     "/tmp/my-queue"
#define MQ_PROJECT_ID   18
#define MESSAGE_TYPE    5

struct message_t {
    long mesg_type;
    int x, y;
};

key_t key = ftok(MQ_PATHNAME, MQ_PROJECT_ID);
int msgid = msgget(key, 0666 | IPC_CREAT);
if (msgid < 0) {
    // handle error...
}

if (fork() == 0) {  // child process
    struct message_t msg = { MESSAGE_TYPE, 10, 20 };
    msgsnd(msgid, &msg, sizeof(msg), 0);

} else {    // parent process
    struct message_t msg;
    msgrcv(msgid, &msg, sizeof(msg), MESSAGE_TYPE, 0);
    mq_close(msgid);

    printf("x + y = %d\n", msg.x + msg.y);
}
```

Pipes

Unix Unnamed Pipes

- Unidirectional, synchronous
- Allow parent-child processes to communicate with each other
- Creating a pipe:
  - `int pipe(int fd[2])`
  - Pass it an array of two integers
  - On success, zero is returned, `fd` contains two file descriptors
    - Use `read()`, `write()`, `close()` on these file descriptors
  - On error, `-1` is returned
    - `errno` is set appropriately
    - `perror()` produces a message on standard error output, describing last error encountered during a system call or library function



Example

```c
int fd[2];
if (pipe(fd) < 0) {
    // handle error...
}

if (fork() == 0) {  // child process
    int x = 10, y = 20;
    write(fd[1], &x, sizeof(x));
    write(fd[1], &y, sizeof(y));
    close(fd[1]);

} else {    // parent process
    int x, y;
    read(fd[0], &x, sizeof(x));
    read(fd[0], &y, sizeof(y));
    close(fd[0]);

    printf("x + y = %d\n", x + y);
}
```

Named Pipes (aka FIFO)

- A named pipe is a first-in-first-out queue
  - Written by one process and read by another
  - Possible to be attached to multiple processes
- An extension to the traditional pipe concept on Unix which lasts only as long as the process
  - A named pipe can last beyond the life of the process
  - A named pipe appears as a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from it, in the same way as an ordinary file

Example

```c
#define FIFO_PATH "/tmp/my-fifo"

if (mkfifo(FIFO_PATH, 0666) < 0) {
    // handle error...
}

if (fork() == 0) {  // child process
    int x = 10, y = 20;
    int fd = open(FIFO_PATH, O_WRONLY);
    write(fd, &x, sizeof(x));
    write(fd, &y, sizeof(y));
    close(fd);

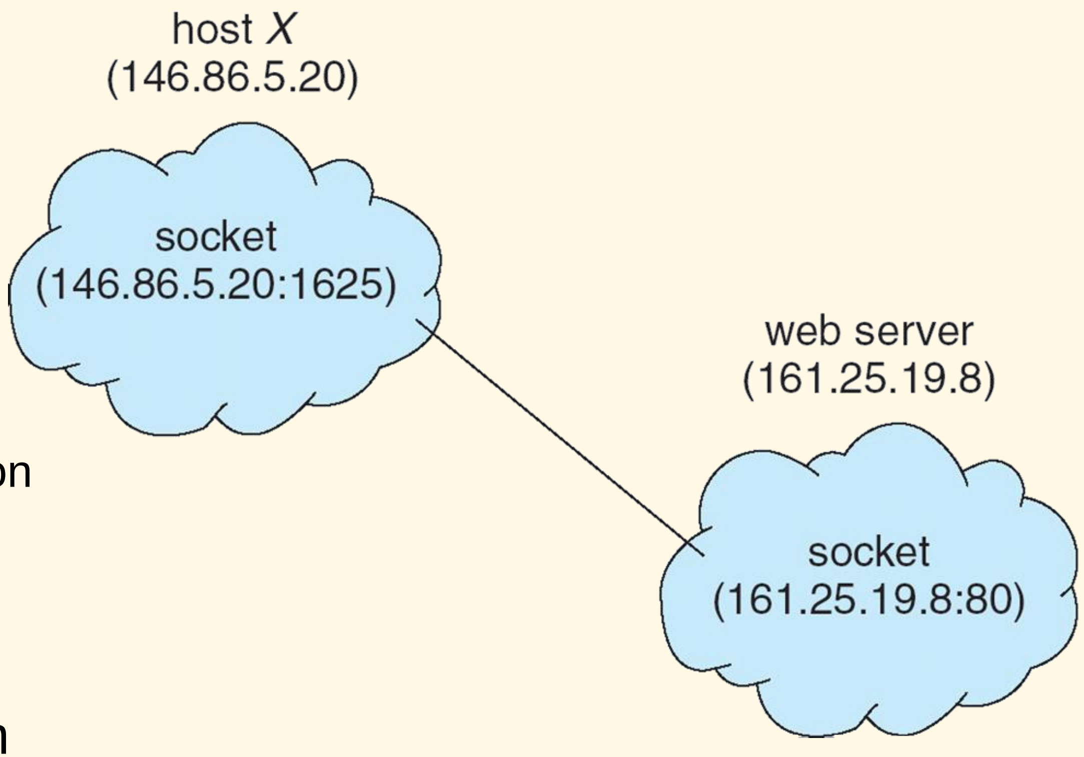
} else {    // parent process
    int x, y;
    int fd = open(FIFO_PATH, O_RDONLY);
    read(fd, &x, sizeof(x));
    read(fd, &y, sizeof(y));
    close(fd);

    printf("x + y = %d\n", x + y);
}
```

Sockets

Sockets

- A socket is defined as an endpoint for communication on the Internet
  - Identified by IP address and port
  - Networking protocol routes the packet to the destined host
  - Different servers listens for requests on well known port
    - E.g., the socket 146.86.5.20:1625 refers to port 1625 on host 146.86.5.20
- Communication consists between a pair of sockets
  - TCP socket: reliable, connection oriented, stream based – more popular method
  - UDP socket: unreliable, connectionless, datagram based



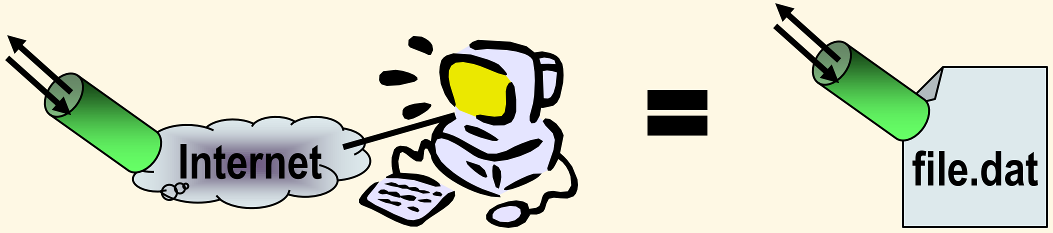
Client/Server Model

- Form of communication used by all network applications
- One application initiates communication and the other accepts

Server		**Client**
• Starts first • Passively waits for contact from a client at a prearranged location • Responds to requests, can handle multiple remote clients simultaneously	⟷	• Starts second • Actively contacts a server with a request • Waits for response from server

The Socket Interface

- Berkeley Sockets API
  - Originally developed as part of BSD Unix (under gov't grant)
    - BSD = Berkeley Software Distribution
  - Now the most popular C/C++ API for writing applications over TCP/IP
    - Also emulated in other languages: Perl, Tcl/Tk, etc.
    - Also emulated on other operating systems: Windows, etc.
- Basic usage:



  - TCP: a socket is like a file
    - servers (passive open) `listen()` and `accept()`
    - clients (active open) `connect()`
    - both sides can then `read()` and/or `write()` (or `send()` and `recv()`)
    - then each side must `close()`
  - UDP: uses `sendto()` and `recvfrom()`

TCP

Server

1. Create transport endpoint for incoming connection request: `socket()`
2. Assign transport endpoint an address: `bind()`
3. Announce willing to accept connections: `listen()`
4. Block and wait for incoming request: `accept()`
5. Wait for a packet to arrive: `recv()`
6. Formulate reply (if any) and send: `send()`
7. Release transport endpoint: `close()`

Client

1. Create transport endpoint: `socket()`
2. Assign transport endpoint an address (optional): `bind()`
3. Determine address of server
4. Connect to server: `connect()`
5. Formulate message and send: `send()`
6. Wait for packet to arrive: `recv()`
7. Release transport endpoint: `close()`

UDP

Server

1. Create transport endpoint for incoming connection request: `socket()`
2. Assign transport endpoint an address: `bind()`
3. Wait for a packet to arrive: `recvfrom()`
4. Formulate reply (if any) and send: `sendto()`
5. Release transport endpoint: `close()`

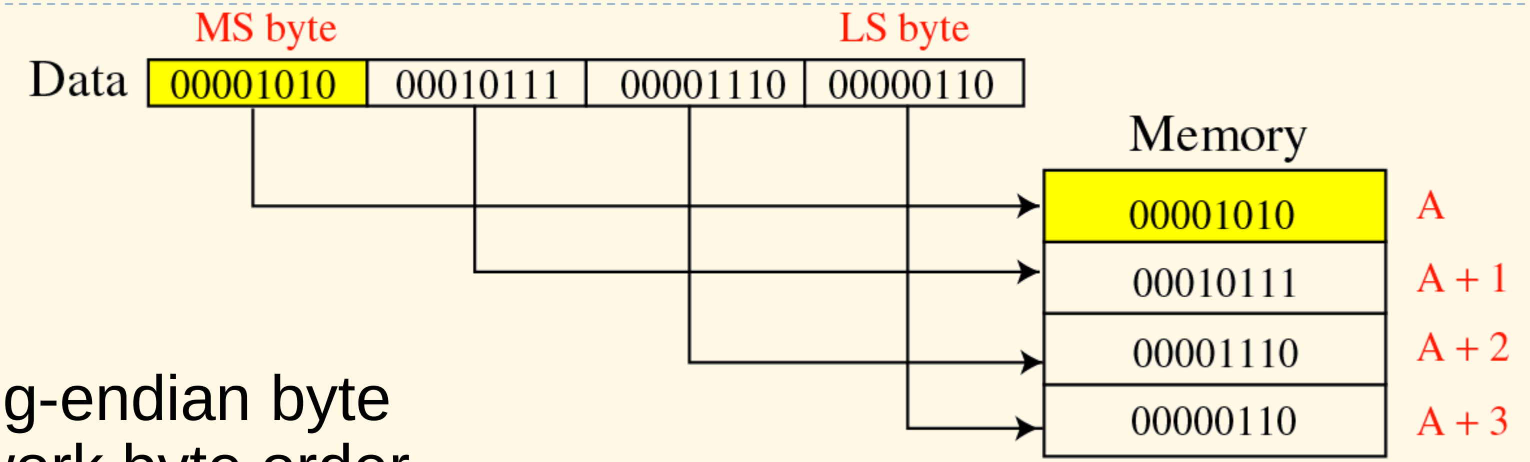
Client

1. Create transport endpoint: `socket()`
2. Assign transport endpoint an address (optional): `bind()`
3. Determine address of server
4. Formulate message and send: `sendto()`
5. Wait for packet to arrive: `recvfrom()`
6. Release transport endpoint: `close()`

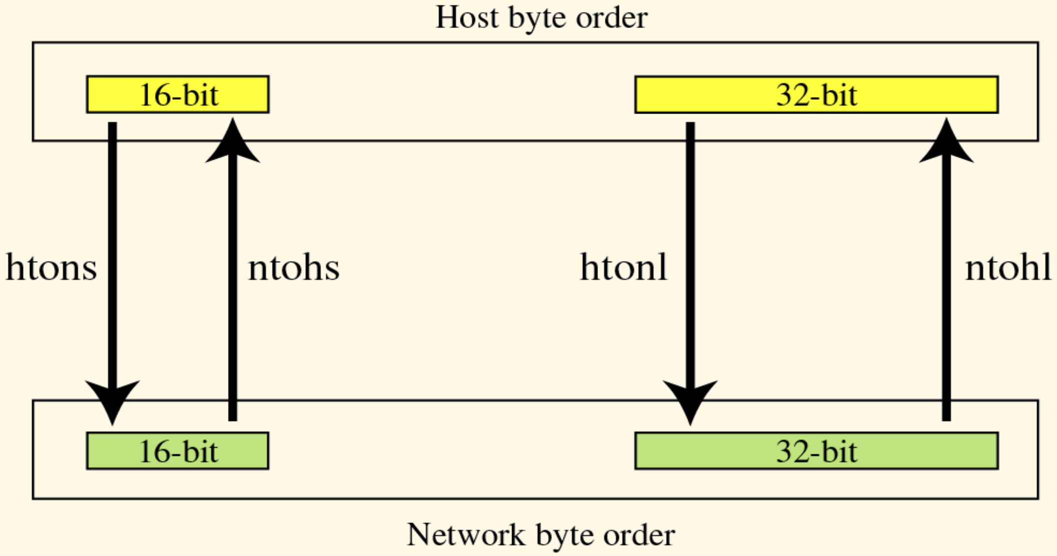
Address Resolution

- ```
  int getaddrinfo(
      const char* host,
      const char* service,
      const struct addrinfo* hints,
      struct addrinfo** res
  );
  ```
- Returns a linked list of one or more `addrinfo` structures, each of which contains an Internet address that can be specified in a call to `bind()` or `connect()`
- IPv4 and IPv6 compatible
  ```

Byte Ordering and Transformation



- TCP/IP uses big-endian byte order, aka network byte order
- Host ⬄ network byte order conversion is necessary
  - `uint16_t htons(uint16_t v)`
  - `uint16_t ntohs(uint16_t v)`
  - `uint32_t htonl(uint32_t v)`
  - `uint32_t ntohl(uint32_t v)`



Example

- See:
  - `tcp_server.c`
  - `tcp_client.c`