CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2016 Lecture 9

Slides based on

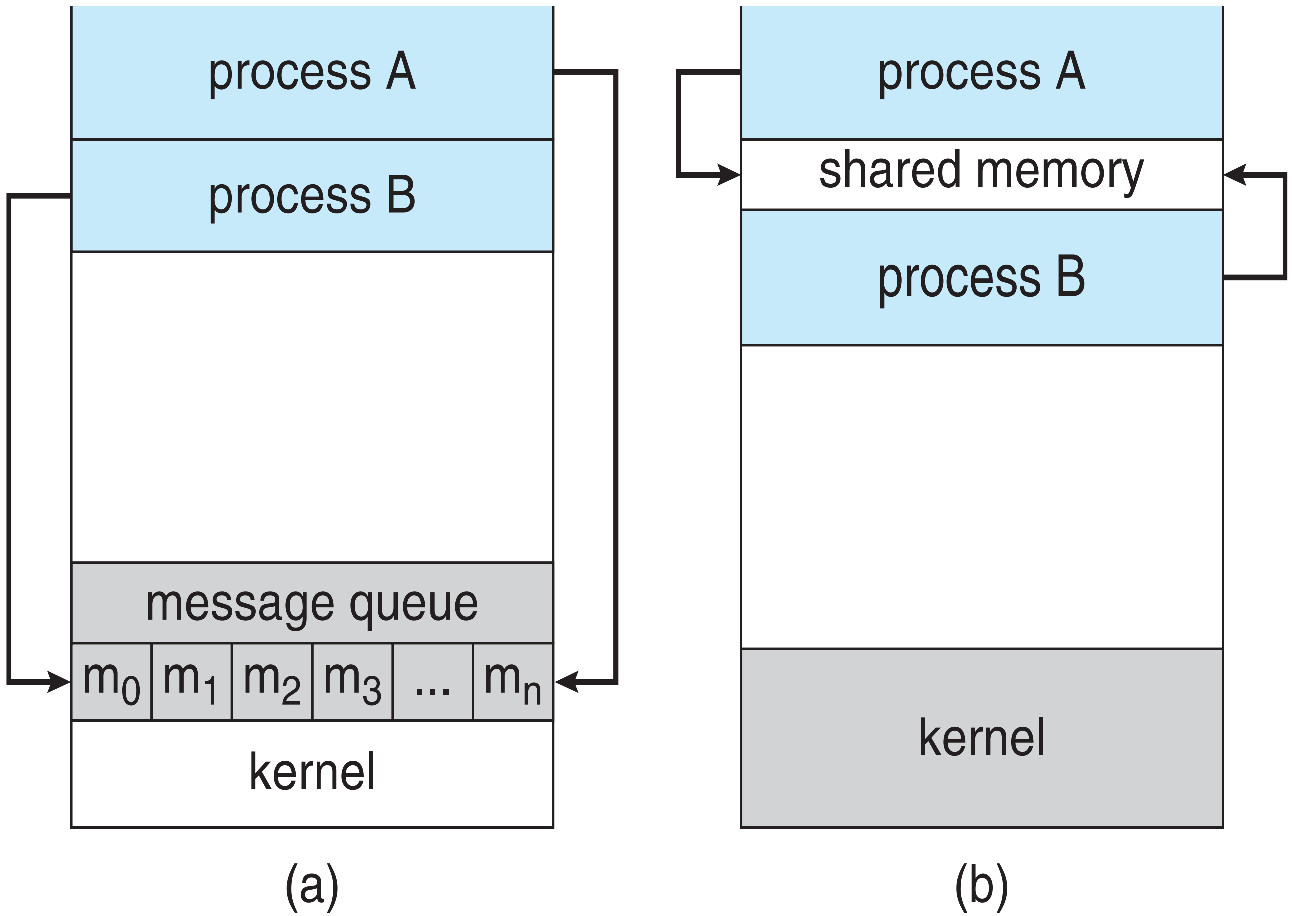
- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

FAQ

- Why would users want to use fork(), wait() etc in your own code, since OS does a good job?
- Is the child [initially] an exact copy of the parent? Does it have the same values in heap, stack, and program counter?
- Zombie: process has exited, but parent hasn't called wait( ) to obtain exit status. Two bytes.
- Process Group ID: a process can send a signal to all members of the PGID.
- Are unbounded buffers used in any systems?
- Why examples use producers and consumers in an infinite loop?

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  - Synchronization is discussed in great details in Chapter 5.
- Example soon.

Only one process may access shared memory at a time

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical: Options (details next)
    - Direct (process to process) or indirect (mail box)
    - Synchronous (blocking) or asynchronous (non-blocking)
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
  - May be owned by a process (to receive) or the OS
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Possible Solutions
  - a link to be associated with at most two processes
  - only one process at a time to execute a receive operation
  - system to select arbitrarily the receiver. (Sender is notified who the receiver was.)

Synchronization*(blocking or not)*

- Message passing may be either blocking or non-blocking
- **Blocking** is termed **synchronous**
  - **Blocking send** -- sender is blocked until message is received
  - **Blocking receive** -- receiver is blocked until a message is available
- **Non-blocking** is termed **asynchronous**
  - **Non-blocking send** -- sender sends message and continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous.**
  - Producer-Consumer problem: Easy both block

Colorado State University

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages Sender must wait if queue full
  3. Unbounded capacity – infinite length Sender never waits

Examples of IPC Systems - POSIX

- Older scheme (System V) using shmget(), shmat(), shmdt(), shmctl()
- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```

    - Returns file descriptor (int) which identifies the file
  - Also used to open an existing segment to share it
  - Set the size of the object

    ```
    ftruncate(shm_fd, 4096);
    ```

  - map the shared memory segment in the address space of the process

    ```
    ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE,
    MAP_SHARED, shm_fd, 0);
    ```

  - Now the process could write to the shared memory

    ```
    sprintf(ptr, "Writing to shared memory");
    ```

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Other process opens shared memory object name
    `shm_fd = shm_open(name, O_RDONLY, 0666);`
    - **Returns file descriptor (int) which identifies the file**
  - map the shared memory object

```
ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED,
shm_fd, 0);
```

  - Now the process can read from to the shared memory object
    `printf("%s", (char *)ptr);`
  - remove the shared memory object
    `shm_unlink(name);`

Colorado State University

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

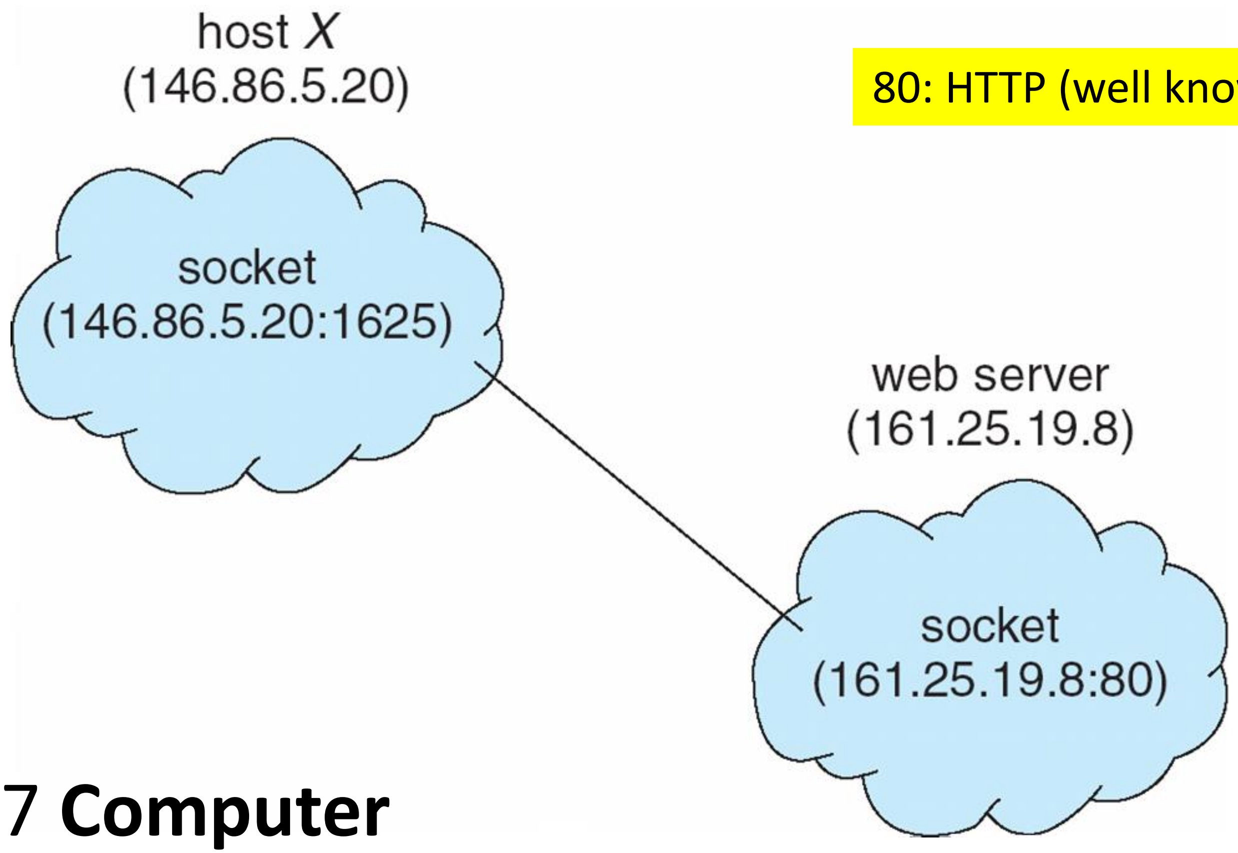
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



80: HTTP (well known)

- CS457 **Computer Networks and the Internet**

Pipes

- Acts as a conduit allowing two processes to communicate
- One of the first IPC implementation mechanisms

Pipes

- Conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - If bidirectional, is it **half-duplex** (one way at a time) or **full-duplex** (both directions simultaneously)?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Pipes

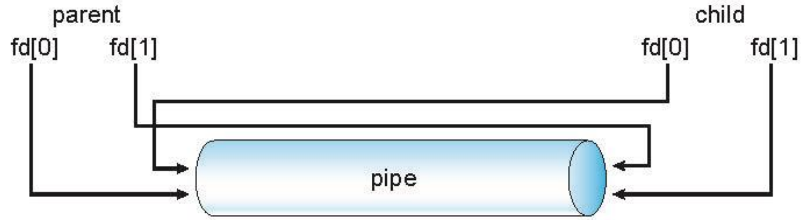
- Command line:
  - Set up pipe between commands

    ls | more

    Output of ls delivered as input to more
- **Ordinary ("anonymous") pipes** –Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Cannot be accessed from outside the process that created it.
- **Named pipes ("FIFO")** – can be accessed without a parent-child relationship.

Ordinary Pipes

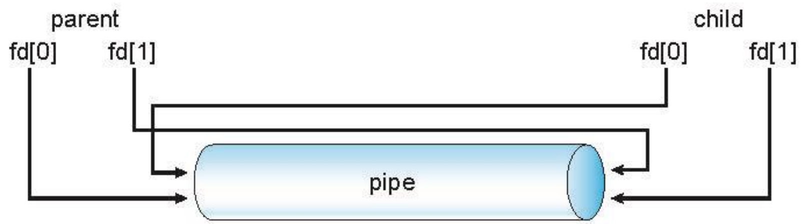
- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional (half duplex)
- **Require parent-child relationship** between communicating processes
- pipe (int fd[]) to create pipe, fd[0] is the read-end, fd[1] is the write-end



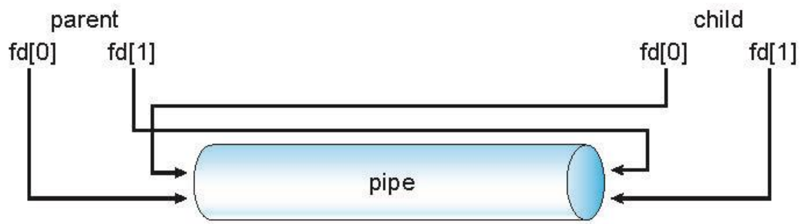
- Windows calls these **anonymous pipes**

Ordinary Pipes

- Pipe is a special type of file.
- Inherited by the child
- Must close unused portions of the the pipe



UNIX pipe example




```
#define READ_END    0
#define WRITE_END   1

        int fd[2];
```

create the pipe:

```
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
```

fork a child process:

```
        pid = fork();
```

parent process:

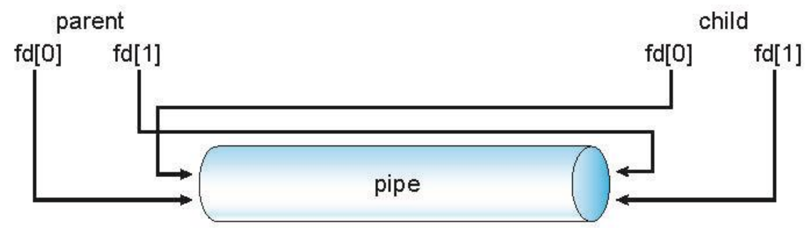
```
                /* close the unused end of the pipe */
                close(fd[READ_END]);

                /* write to the pipe */
                write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

                /* close the write end of the pipe */
                close(fd[WRITE_END]);
```

Colorado State University

UNIX pipe example



child process:

```
/* close the unused end of the pipe */
close(fd[WRITE_END]);

/* read from the pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);
printf("child read %s\n",read_msg);

/* close the write end of the pipe */
close(fd[READ_END]);
```

Named Pipes

- Named Pipes (termed FIFO) are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems