CS 3113 Introduction to Operating Systems

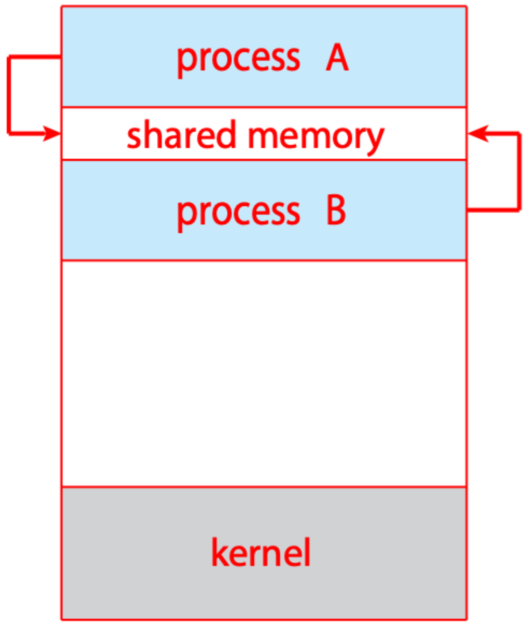
Topic #3. Processes

Outline

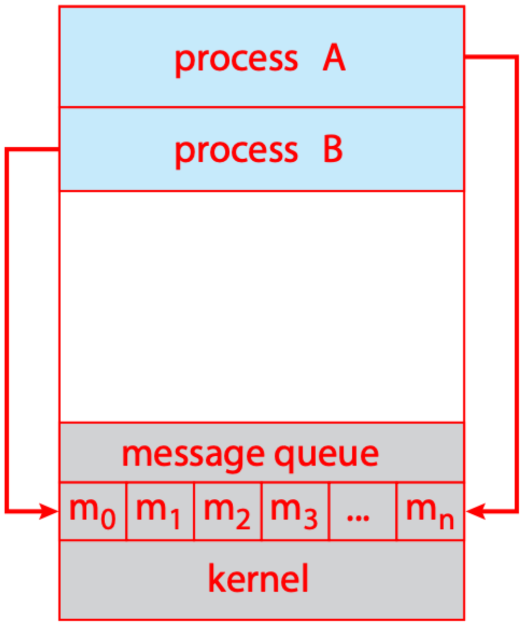
- 3.5 IPC in Shared-Memory Systems
- 3.6 IPC in Message-Passing Systems
- 3.7 Examples pf IPC Systems
- 3.8 Communication in Client–Server Systems

Communication Models

Shared Memory **Message Passing**



(a)



(b)

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of these processes & not the operating system.
    - Good for efficiency (no system calls to update memory)
    - Lots of opportunities for bugs
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
    - Resides in the address space of the process creating the shared memory
    - Other processes must attach the shared memory to their address space

IPC – Shared Memory (Cont'd)

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Producer-Consumer Problem

- Producer: process generates data through some mechanism
- Consumer: process uses data generated by another

Producer-Consumer Problem

Typical approach: implement a data buffer from the producer to the consumer

- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

Circular/Shared Buffer of Items

- Items are instances of type **item**

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- in = the next free location to place a new item
- out = the next full location to remove an item from
- Both the producer and consumer processes have access to this buffer

Circular/Shared Buffer of Items

- in == out: no items in the buffer
- (in+1)%BUFFER_SIZE == out: buffer is full

Circular Buffer: Producer

```
item next_produced;
while (true) {
      // Generate new item
      next_produced = …

      // Wait for there to be space in the buffer
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */

      // Place item in the buffer
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Circular Buffer: Consumer

```
item next_consumed;
while (true) {
	// Wait for item to be available
	while (in == out)
		; /* do nothing */

	// Get the next item
	next_consumed = buffer[out];
	out = (out + 1) % BUFFER_SIZE;

	// Do something with the item
}
```

Outline

- 3.5 IPC in Shared-Memory Systems
- 3.6 IPC in Message-Passing Systems
- 3.7 Examples pf IPC Systems
- 3.8 Communication in Client–Server Systems

IPC in Message Passing Systems

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 generic operations:
  - send(message)
  - receive(message)
- The message size can be either fixed or variable

Message Passing

- 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? (buffer)
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing

Implementation of a communication link

- Physical choices:
  - Shared memory
  - Hardware bus
  - Network
- Logical choices:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication Model

- 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 Model

- Messages are directed to and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- 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 Model

- 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

Mailbox Sharing

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

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send: the sender is blocked until the message is received
  - Blocking receive: the receiver is blocked until a message is available

Synchronization

- Non-blocking is considered asynchronous
  - Non-blocking send: the sender sends the message and continues
    - The message is placed into a temporary buffer
  - Non-blocking receive: the receiver receives:
    - A valid message, or
    - Null message (nothing to receive)
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization with Rendezvous

Producer-consumer becomes trivial

```
message next_produced;
while (true) {
      /* produce an item in next produced */
        send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link
- Implemented in one of three ways:
  - Zero capacity: no messages are queued on a link. Sender must wait for receiver (rendezvous)
  - Bounded capacity: finite length of n messages
    - Sender must wait if link full
    - Receiver must wait if link has nothing
    - This is our circular buffer example!
  - Unbounded capacity: infinite length Sender never waits

Buffering (Cont’d)

Advantages:

- Processes don’t have to wait for each other
- -> Potentially more efficient use of CPU and other resources

Challenges:

- Capacity must be large enough
- Consumer can potentially fall far behind
  - If timing is important, this can be a big issue

Quiz

- What are advantages and disadvantages of an unbounded capacity buffer?

Quiz answer

- Advantages: the operating-system provides the buffers in the form of a queue with indefinite length. This means that any number of messages can wait in the queue, so the sender will never have to block

- The disadvantage of automatic buffering is that it involves a more complex system that may be difficult to implement so that it manages memory efficiently. A scheme may reserve a sufficiently large memory space for the messages; however the memory may never fully be used, thereby wasting memory space.

Outline

- 3.5 IPC in Shared-Memory Systems
- 3.6 IPC in Message-Passing Systems
- 3.7 Examples pf IPC Systems
- 3.8 Communication in Client–Server Systems

Shared Memory in POSIX (includes Linux)

- Process first creates shared memory segment (or opens an existing one):

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Set the size of the object:

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

- Create a pointer to the shared memory:

```
shared_memory = mmap(shm_fd, 0, 4096 PROT_WRITE,
                                  MAP_SHARED, shm_fd, 0);
```

- Now the process can write to the shared memory

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

Shared Memory 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;
}
```

Shared Memory 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;
}
```

Pipes

- Act as a conduit allowing two processes on the same computer to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Types of Pipes

- Ordinary pipes: cannot be accessed from outside the process that created it.
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes: can be accessed without a parent-child relationship.

Ordinary Pipes Notes

- Parent creates the process and then forks()
- Both parent and child close one half of the pipe so there is only one reader and one writer for the pipe
  - Either the parent or the child can become the producer
- When the producer closes the fd, the bytes in the buffer are still available for reading
  - When there are no more bytes, the consumer will receive an EOF marker

Named Pipes Notes

- One process opens for reading, the other for closing
- open() system call blocks until the other process also calls open()
- When the producer closes, the bytes in the pipe's buffer are still available for reading
  - When there are no more bytes, the consumer receives an EOF marker
- If you wrap a FILE* around a pipe fd, then make sure that you flush the FILE's buffer (different than the pipe buffer) to make sure that the bytes are actually sent

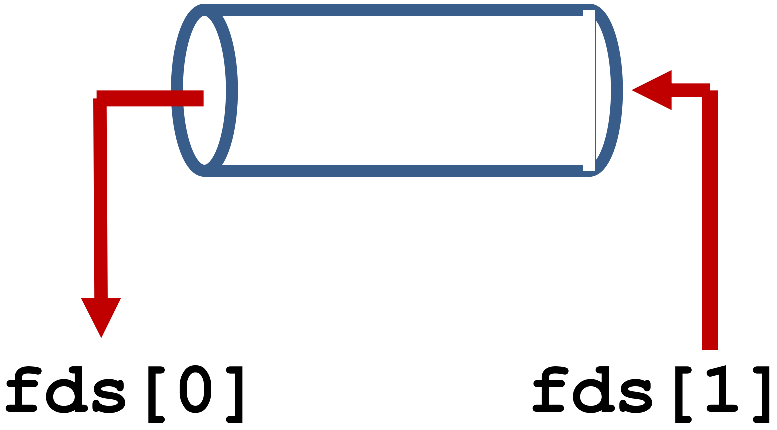
Ordinary Pipes

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

Ordinary Pipe

```
int fds[2];
int ret = pipe(fds);
if(ret < 0) exit(-1);
/* Now use the pipe */
```



- fds[0]: output from pipe
- fds[1]: input to pipe

Ordinary Pipes for Communication

- The pipe is implemented inside the kernel (so, it does not exist within the process)
- However, the process maintains this pair of file descriptors, which allow it to reference the pipe

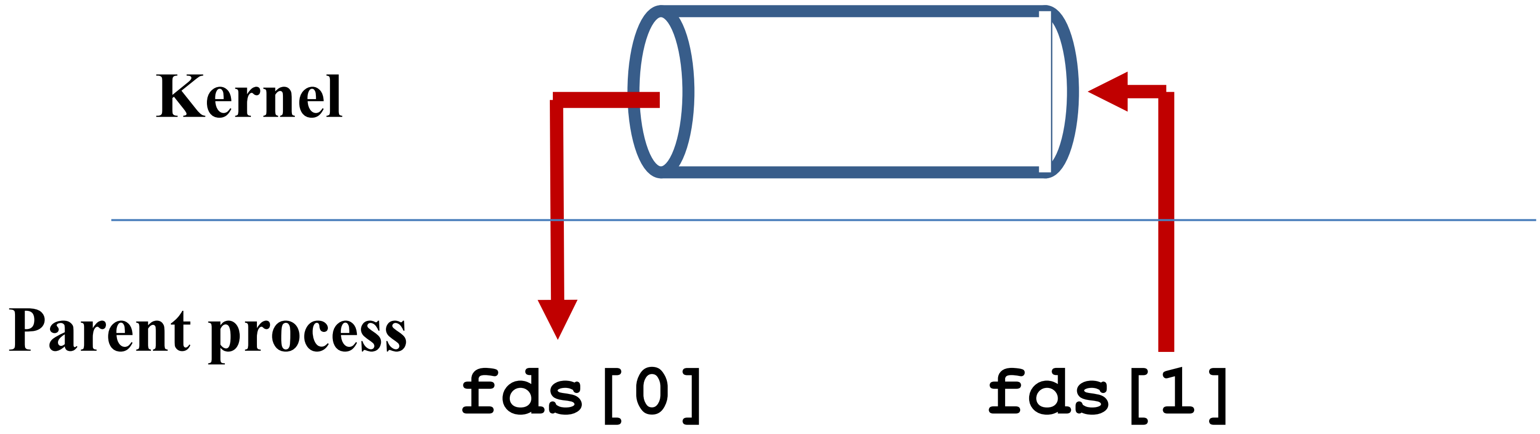
Ordinary Pipes for Communication

- The file descriptors cannot be shared outside of the process
- But: if the process forks(), then both the parent and child will have copies of the file descriptors!
  - And these reference the same pipe

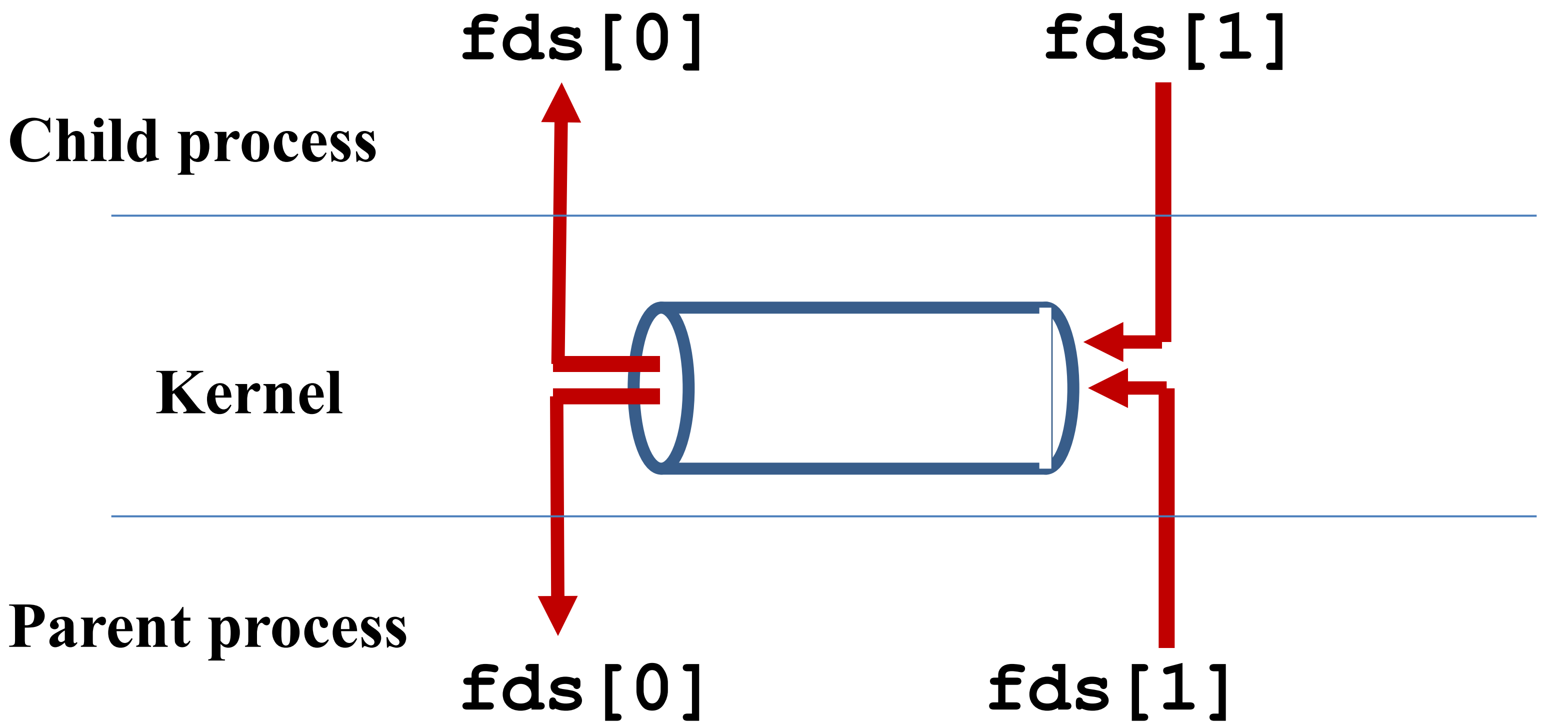
Ordinary Pipe with Fork()

```
int fds[2];
int ret = pipe(fds);
if(ret < 0) exit(-1);
/* Now use the pipe */
int pid = fork()
if(pid > 0) {                /* Note: leaving off error case*/
        /* parent code */
}else {
        /* child code */
}
```

Before Fork



After Fork

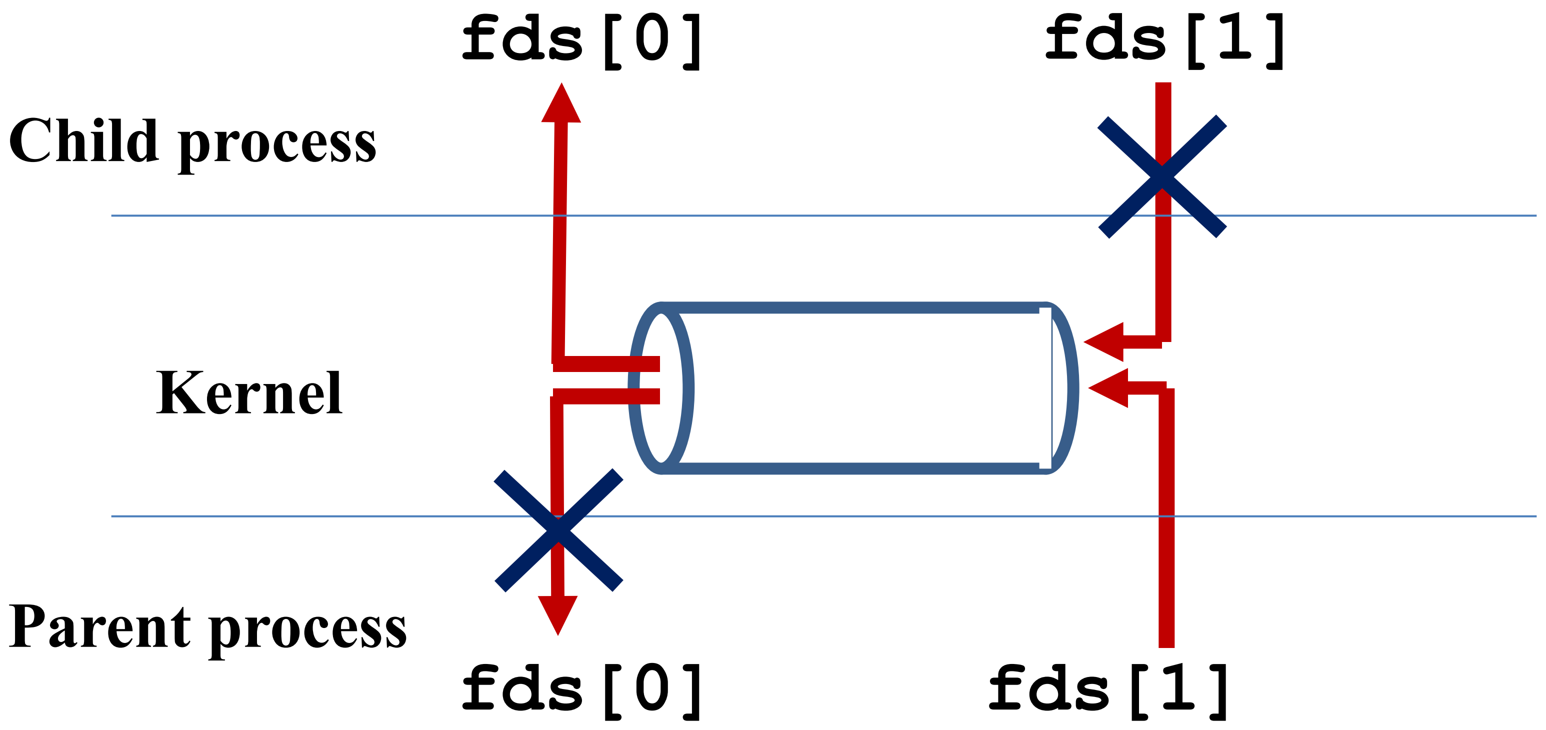


Pipes

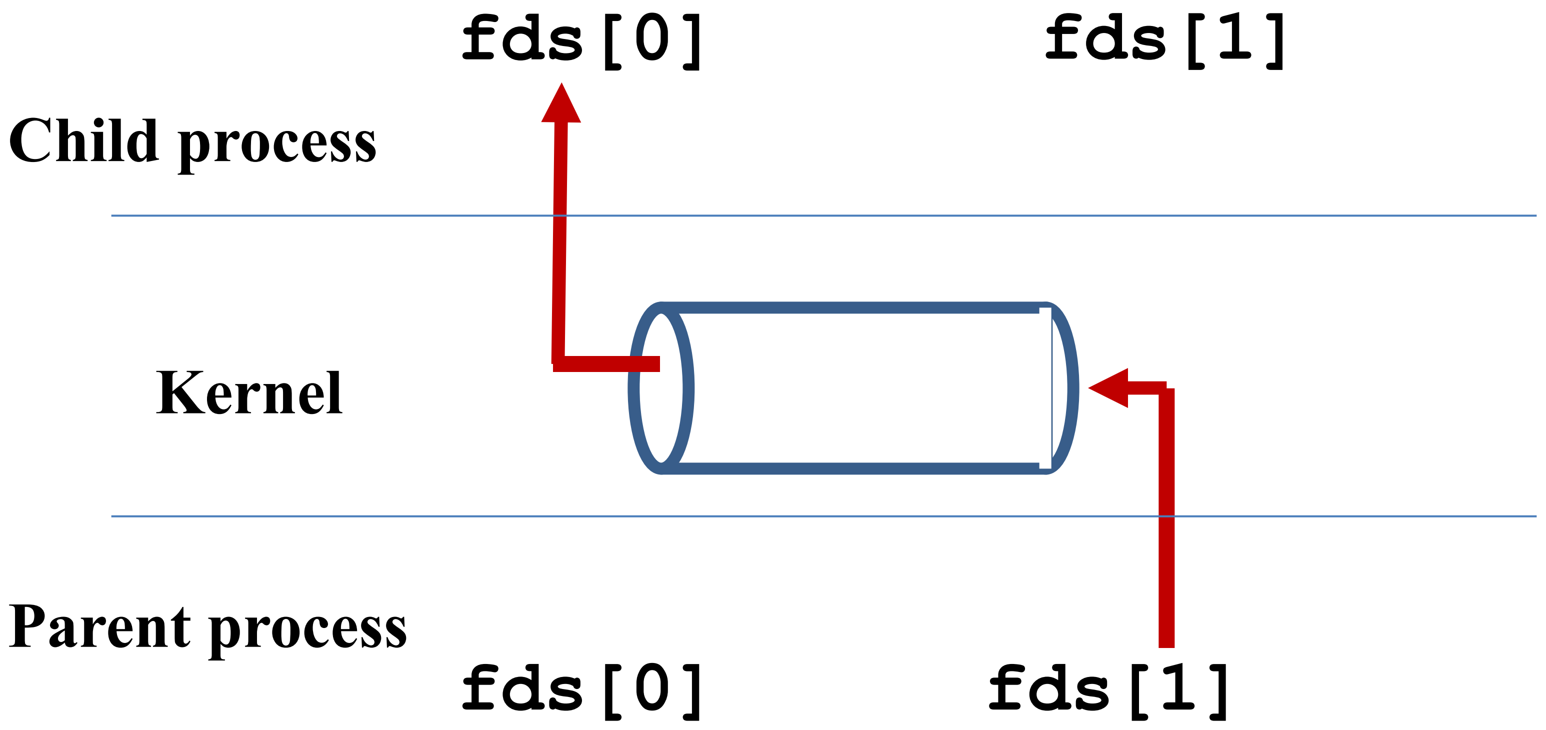
For our purposes, we will assume:

- Pipes will only work properly with one reader and one writer (this is the typical use case)
- This means, after the fork, each of the parent and child will close one of the two file descriptors

After Fork: Parent is the Producer



After Fork: Parent is the Producer



After Fork: Parent is the Producer

```
// Now use the pipe
int pid = fork()
if(pid > 0) {                    // Note: leaving off error case
        // parent code
        close(fds[0])
                :
}else {
        // child code
        close(fds[1])
                :
}
```

After Fork: Parent is the Producer

After fork and closing:

- Parent can write bytes to fds[1]
- Child can read these bytes from fds[0]

After Fork: Parent is the Producer

- The pipe has a buffer: it will hold written bytes until they are read
- If the writer closes the pipe, then
  - The reader gets to read the remaining bytes
  - But then will see an EOF

- Windows calls ordinary pipes ***anonymous pipes***

Named Pipes

Named Pipes are more powerful than ordinary pipes

- Communication ***can*** be 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

Named Pipes in Unix

- Access points exist in the file system
  - Open them just as you would a file!
  - Use read()/write() to receive/send data
- Can have multiple readers/writers
  - A message is delivered to one randomly selected reader
  - So, effectively, they are bidirectional
  - **However, we will use them as unidirectional pipes**
- Create at the command line (or programmatically):

`mkfifo [NAME]`

Outline

- 3.5 IPC in Shared-Memory Systems
- 3.6 IPC in Message-Passing Systems
- 3.7 Examples pf IPC Systems
- 3.8 Communication in Client-Server Systems

Client-Server Model of Communication

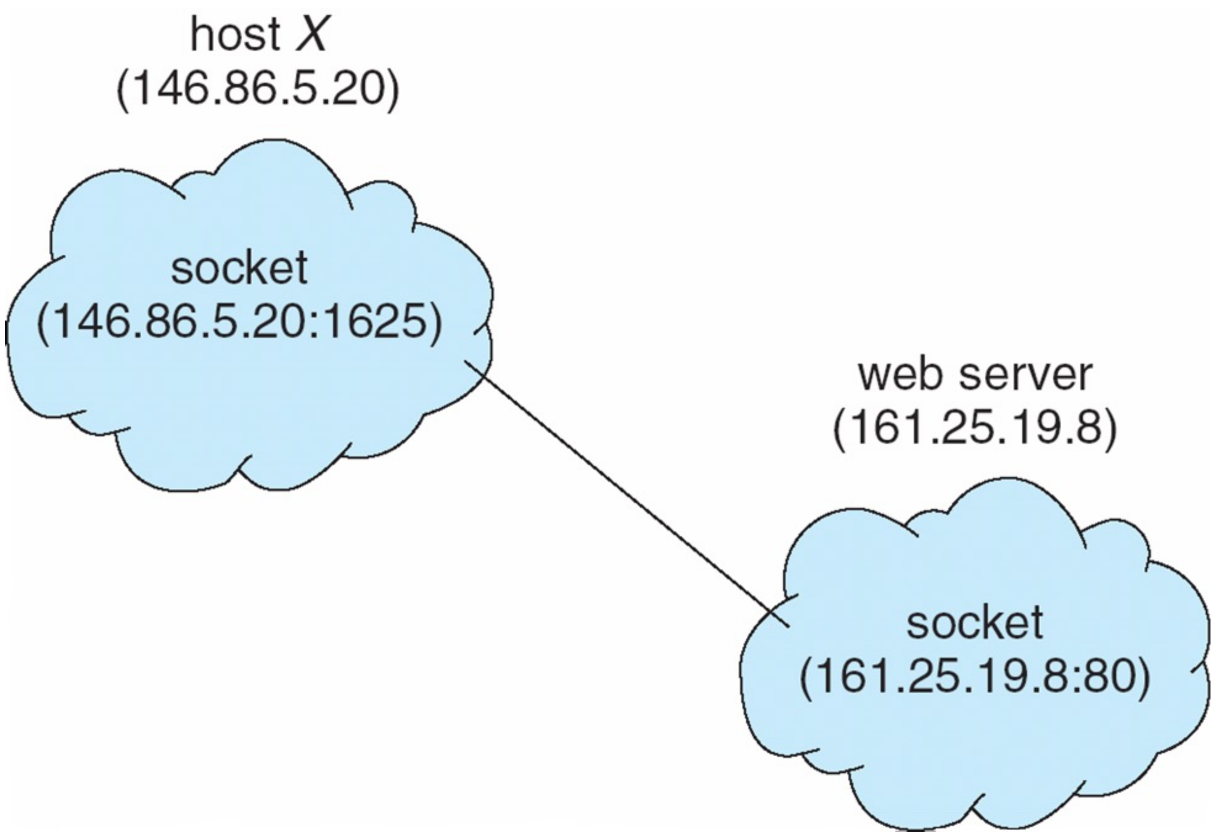
- Built on top of Producer/Consumer concept
- Server: provides some service (data, computation)
- Client: requests actions on the part of the server
- Implementation choices include:
  - Sockets
  - Pipes
  - Remote Procedure Calls
  - Remote Method Invocation (Java)

Sockets

A socket is defined as an endpoint for communication

- Identified by a 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
- All communication is done between a pair of sockets (one for client; the other for server)
- All ports below 1024 are well known and are used for standard services. Others are available for users
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running

Socket Communication



Remote Procedure Calls (RPCs)

- From the programmer's perspective, they appear as functions/methods that take arguments and return a value
- Under the hood, this function call:
  - Contacts a server
  - Sends the arguments to the server
  - Server does the work and sends the result back
  - Return the return value back to the client

Procedure for Using Shared Memory

- Find a *key*: Unix uses this key for identifying shared memory segments
- Use shmget() to allocate a shared memory
- Use shmat() to attach a shared memory to an address space
- Use shmdt() to detach a shared memory from an address space
- Use shmctl() to to deallocate a shared memory

Project 1

- Header (incomplete)

```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <errno.h>


/* key number */
#define SHMKEY ((key_t) 1497)
```

Key

- Unix uses this key for identifying shared memory segments.
  - A key is a value of type key_t
  - There are three ways to generate a key
    - Do it yourself
    - Use function ftok()
    - Ask the system to provide a private key

Key

- Do it yourself:
  - #define SHMKEY ((key_t) 1497)
- Use ftok() to generate one for you
  - key_t = ftok(char *path, int ID);
  - path is a path name (e.g., “./”)
  - ID is an integer (e.g., ‘a’)
  - Function ftok() returns a key of type key_t: Key = ftok(“./”, ‘x’)
- Keys are global entities. If other processes know your key, they can access your shared memory.
- Ask the system to provide a private key using IPC_PRIVATE

Project 1

- shared memory

```
typedef struct
{
 int value;
} shared_mem;

shared_mem *total;
```

- process1 increases the value of shared variable "total“ * by some number
- process2
- process3
- process4

Project 1

- Main function

```
int   shmid, pid1,pid2, pid3,pid4, ID,status;
char *shmadd;
shmadd = (char *) 0;
```

- /* Create and connect to a shared memory segment*/

```
if ((shmid = shmget (SHMKEY, sizeof(int), IPC_CREAT | 0666)) < 0){
    perror ("shmget");
     exit (1);}
 if ((total = (shared_mem *) shmat (shmid, shmadd, 0)) ==
(shared_mem *) -1) {
      perror ("shmat");
      exit (0);}
```

Project 1

- Initialize shared memory to 0

```
total->value = 0;
```

- Create processes

```
if ((pid1 = fork()) == 0)
    process1();
```

- Create additional processes

Project 1

- Parent wait for child processes to finish and print ID of each child. Three ways:
  - wait(&status)
  - wait(NULL)
  - waitpid(pid1, NULL, 0);
  - printf("Child with pid %d has just exited.\n", pid1);

Project 1

- To detach a shared memory, use shmdt(total);
  - total is the pointer returned by shmat()

```
if (shmdt(total) == -1)   {
    perror ("shmdt");
    exit (-1);
  }
```

- To remove a shared memory, use shmctl(shmid, IPC_RMID, NULL);
  - shmid is the shared memory ID returned by shmget().
- After a shared memory is removed, it no longer exists.

Notes

- If you did not remove your shared memory segments (e.g., program crashes before the execution of shmctl()), they will be in the system forever. This will degrade the system performance.
- Use the ipcs command to check if you have shared memory segments left in the system.
- Use the ipcrm command to remove your shared memory segments.