Operating Systems

Associate Prof. Yongkun Li
中科大-计算机学院 副教授
http://staff.ustc.edu.cn/~ykli

Ch5
Process Communication & Synchronization

Story so far…

- Process concept + operations
  - Programmer’s perspective + kernel’s perspective
- Thread
  - Lightweight process

- We mainly talked about the stuffs related to a single process/thread, what if multiple processes exist…

Processes

- The processes within a system may be
  - ***independent*** or
    - Independent process cannot affect or be affected by other processes
  - ***cooperating***
    - Cooperating process can affect or be affected by other processes

- Note: Any process that shares data with others is a cooperating process

Cooperating Processes

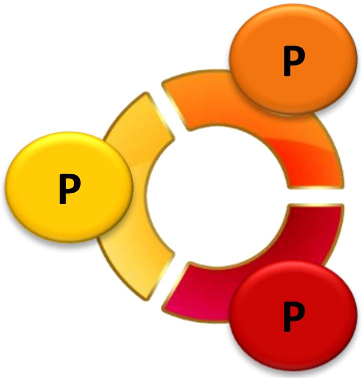
- Why we need cooperating processes
  - Information sharing
    - e.g., shared file
  - Computation speedup
    - executing subtasks in parallel
  - Modularity
    - dividing system functions into separate processes
  - Convenience

Cooperating Processes

- Paradigm for cooperating processes
  - **Producer-consumer problem**, useful metaphor for many applications (abstracted problem model)
    - ***producer*** process produces information that is consumed by a ***consumer*** process
    - At least one producer and one consumer

- Cooperating processes need
  - **interprocess communication (IPC)** for exchanging data

Inter-process communication (IPC)
- What and how?

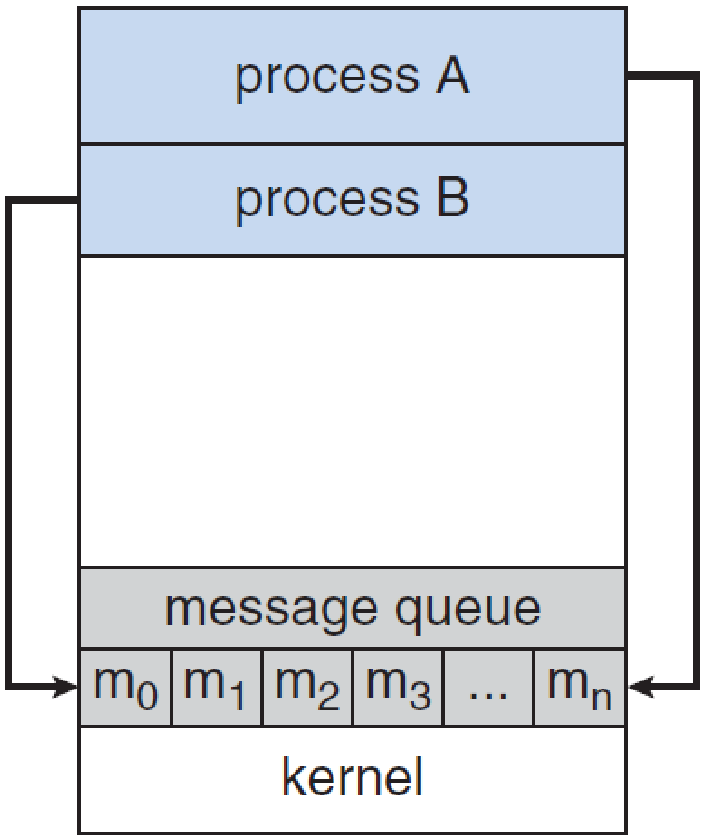


Interprocess Communication

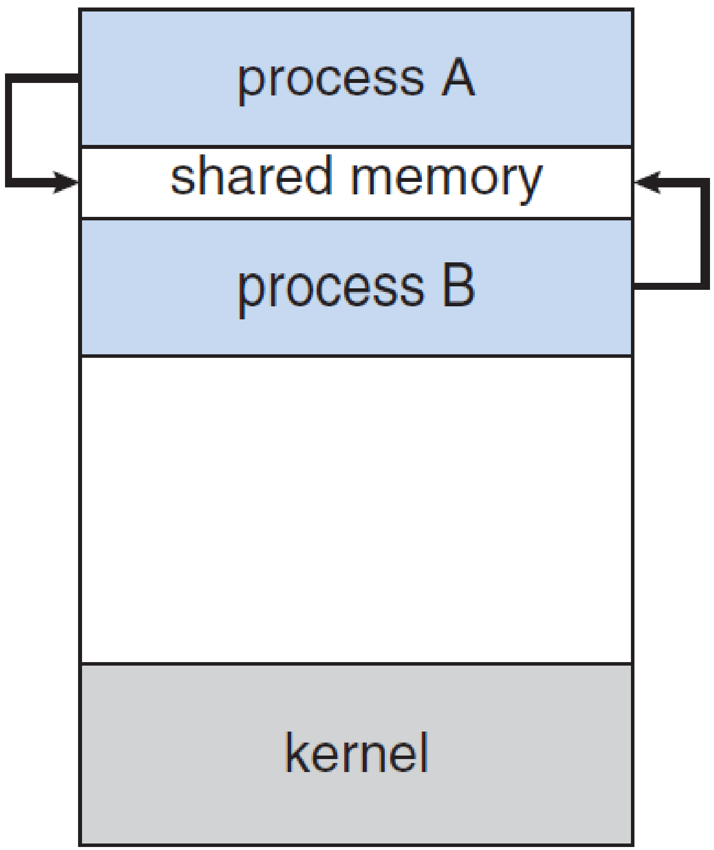
- IPC: used for exchanging data between processes

- Two (abstracted) models of IPC
  - **Shared memory**
    - Establish a shared memory region, read/write to shared region
    - Accesses are treated as routine memory accesses
    - Faster
  - **Message passing**
    - Exchange message
    - Require kernel intervention
    - Easier to implement in distributed system

Communications Models



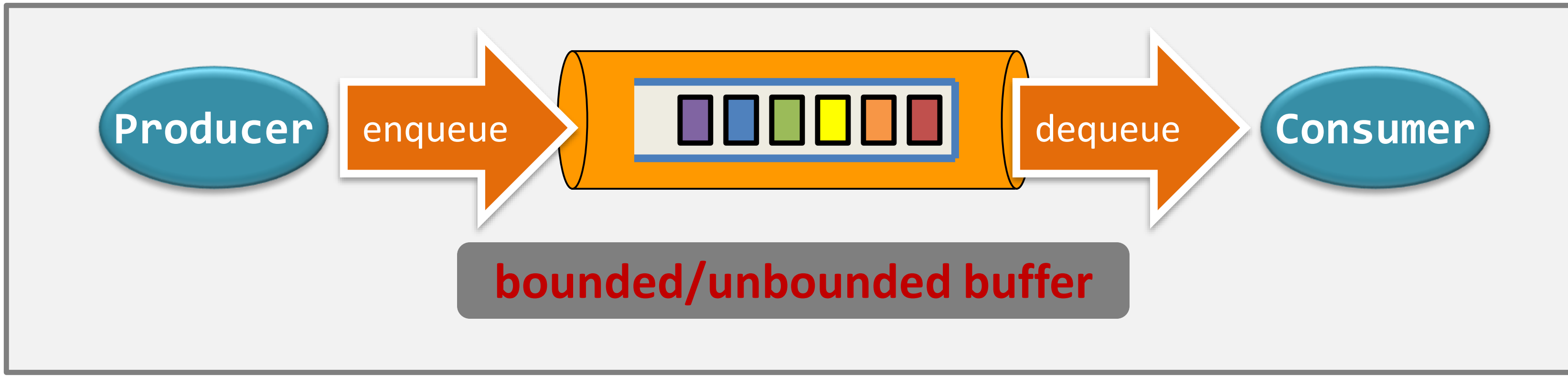
Message passing



Shared memory

Producer-Consumer Problem

- Shared memory solution
  - A buffer is needed to allow processes to run concurrently



A buffer	-It is a shared object; -It is a queue (imagine that it is an array implementation of queue).
A producer process	-It produces a unit of data, and -writes that a piece of data to the tail of the buffer at one time.
A consumer process	-It removes a unit of data from the head of the bounded buffer at one time.

Producer-Consumer Problem

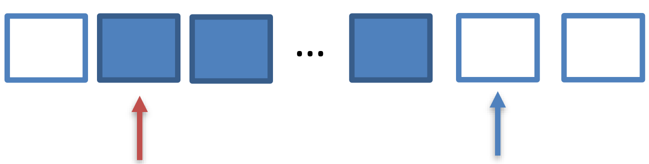
- Focus on bounded buffer: what are the requirements?

Producer-consumer requirement #1	When the **producer** wants to (a) put a new item in the buffer, but (b) **the buffer is already full**… Then, (1) **The producer should be suspended**, and (2) **The consumer should wake the producer up** after she has dequeued an item.
Producer-consumer requirement #2	When the **consumer** wants to (a) consumes an item from the buffer, but (b) **the buffer is empty**… Then, (1) **The consumer should be suspended**, and (2) **The producer should wake the consumer up** after she has enqueued an item.

Producer-consumer solution (shared mem)

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Shared memory by producer & consumer processes



out (consumer)   in (producer)

Only allows BUFFER_SIZE-1 items at the same time. Why?

Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item */
}
```

Message Passing

- Communicating processes may reside on different computers connected by a network
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

- Implementation issues (logical):
  - Naming: Direct/indirect communication
  - Synchronization: Synchronous/asynchronous
  - Buffering

Naming

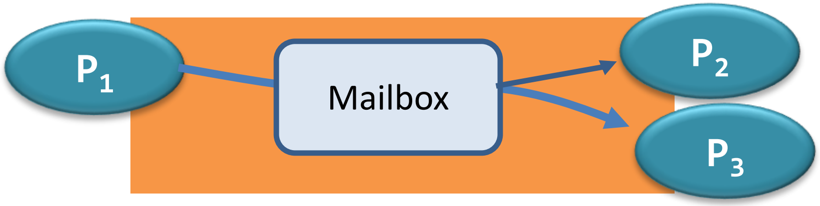
- How to refer to each other?
- Direct communication: explicitly name each other
  - Operations (symmetry)
    - **`send`** (*Q, message*) – send a message to process Q
    - **`receive`**(*P, message*) – receive a message from process P
  - Properties of communication link
    - Links are established automatically (every pair can establish)
    - A link is associated with exactly one pair of processes
    - Between each pair, there exists exactly one link
  - Disadvantage: limited modularity (hard-coding)

Naming

- How to refer to each other?
- Indirect communication: sent to and received from mailboxes (ports)
  - Operations
    - **`send`** (*A, message*) – send a message to mailbox A
    - **`receive`**(*A, message*) – receive a message from mailbox A
  - Properties of communication link
    - A link is established between a pair of processes only if both members have a shared mailbox
    - A link may be associated with more than two processes
    - Between each pair, a number of different links may exist

Issues of Indirect Communication

- ISSUE1: Who receives the message when multiple processes are associated with one link?
  - Who gets the message?



  - Policies
    - 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 (based on an algorithm). Sender is notified who the receiver was.

- ISSUE2: Who owns the mailbox?
  - The process (ownership may be passed)
  - The OS (need a method to create, send/receive, delete)

Synchronization

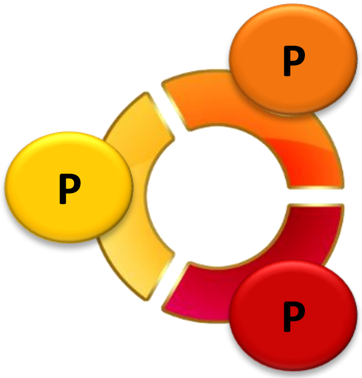
- How to implement send/receive?
  - **Blocking** is considered **synchronous**
    - **Blocking send** - the sender is blocked until the msg is received
    - **Blocking receive** - the receiver is blocked until a msg is available
  - **Non-blocking** is considered **asynchronous**
    - **Non-blocking send** - the sender sends the message and resumes
    - **Non-blocking receive** - the receiver receives a valid msg or null
- Different combinations are possible
  - When both send and receive are blocking, we have a *rendezvous* between the processes.
  - Other combinations need *buffering*.

Buffering

- Messages reside in a temporary queue, which can be implemented in three ways
  - **Zero capacity** – no messages are queued on a link, sender must wait for receiver (no buffering)
  - **Bounded capacity** – finite length of *n* messages, sender must wait if link is full
  - **Unbounded capacity** – infinite length, sender never waits

Inter-process communication (IPC)

- What and how?
- POSIX shared memory



POSIX Shared Memory

- POSIX shared memory is organized using memory-mapped file
  - Associate the region of shared memory with a file
- Illustrate with the producer-consumer problem
  - Producer
  - Consumer

POSIX Shared Memory

- Producer
  - Create a shared-memory object
    - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

Name of the shared memory object

Create the object if it does not exist

Open for reading & writing

Directory permissions

POSIX Shared Memory

- Producer
  - Create a shared-memory object
    - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
  - Configure object size
    - `ftruncate(shm_fd, SIZE);`

File descriptor for the shared mem. Obj.

Size of the shared-memory object

POSIX **Shared** Memory

- Producer
  - Create a shared-memory object
    - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
  - Configure object size
    - `ftruncate(shm_fd, SIZE);`
  - Establish a memory-mapped file containing the object
    - `ptr = mmap(0,SIZE, PROT_WRITE,MAP_SHARED,shm_fd,0);`

Allows writing to the object
(only writing is necessary for producer)

Changes to the shared-memory object will
be visible to all processes sharing the object

POSIX Shared Memory

- Consumer
  - Open the shared-memory object
    - `shm_fd = shm_open(name, O_RDONLY, 0666);`

Open for read only

POSIX Shared Memory

- Consumer
  - Open the shared-memory object
    - `shm_fd = shm_open(name, O_RDONLY, 0666);`
  - Memory map the object
    - `ptr = mmap(0,SIZE, PROT_READ,MAP_SHARED,shm_fd,0);`

Allows reading to the object
(only reading is necessary for consumer)

POSIX Shared Memory

- Consumer
  - Open the shared-memory object
    - `shm_fd = shm_open(name, O_RDONLY, 0666);`
  - Memory map the object
    - `ptr = mmap(0,SIZE, PROT_READ,MAP_SHARED,shm_fd,0);`
  - Remove the shared memory object
    - `shm_unlink(name);`

POSIX Shared Memory – Complete Solution

Producer

```c
#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;
}
```

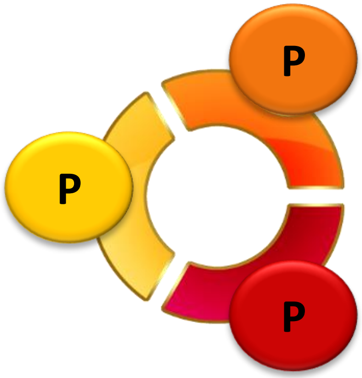
Consumer

```c
#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;
}
```

Direct access to the shared memory region

Inter-process communication (IPC)

- What and how?
- POSIX shared memory
- Sockets

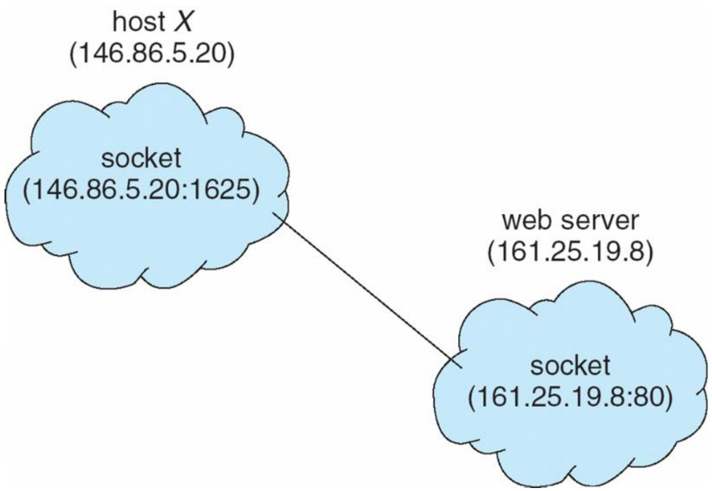


Sockets

- A **socket** is defined as an endpoint for communication (over a network)
  - A pair of processes employ a pair of sockets
  - A socket is identified by an **IP address** and a **port** number
  - All ports below 1024 are used for standard services
    - telnet server listens to port 23
    - FTP server listens to port 21
    - HTTP server listens to port 80

Sockets

- Socket uses a client-server architecture
  - ➢ Server waits for incoming client requests by listening to a specific port
  - ➢ Accepts a connection from the client socket to complete the connection



- All connections must be unique
  - Establishing a new connection on the same host needs another port (>1024)
- Special IP address 127.0.0.1 (**loopback**) refers to itself
  - Allow a client and server on the same host to communicate using the TCP/IP protocol

Example in Java

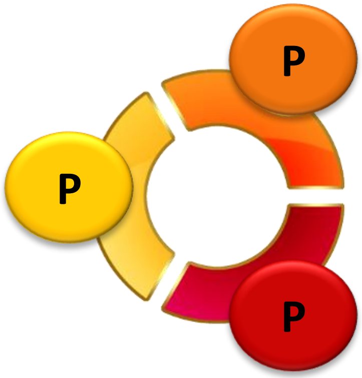
- Three types of sockets
  - **Connection-oriented** (**TCP**), **Connectionless** (**UDP**), **Multicast** – data can be sent to multiple recipients

```java
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

```java
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

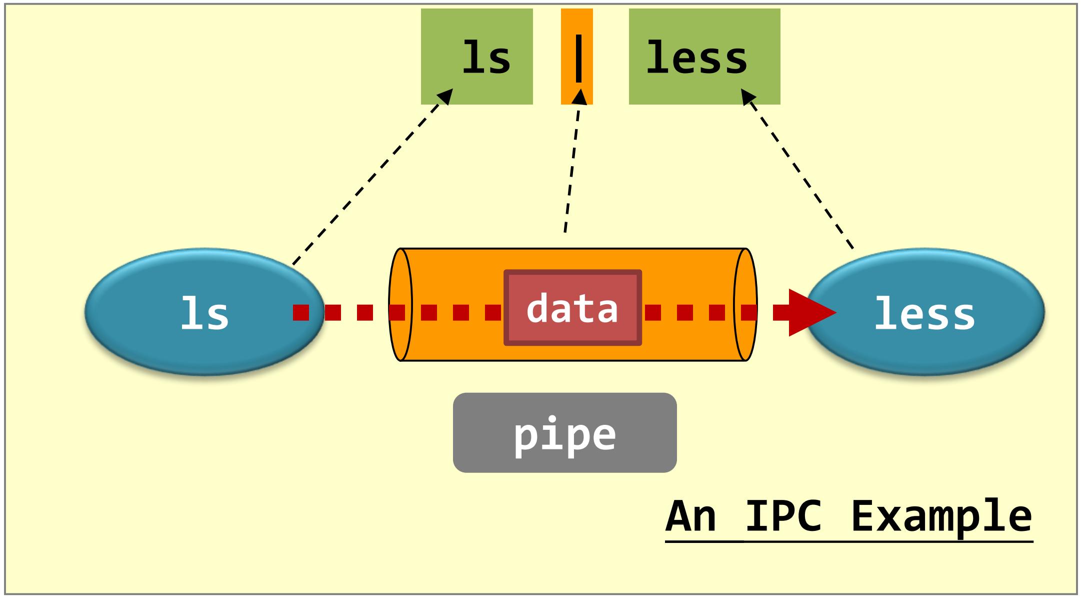
Inter-process communication (IPC)

- What and how?
- POSIX shared memory
- Sockets
- Pipes



What is pipe?

- Pipe is a **shared object.**
  - **Using pipe** is a way to realize IPC.
  - Acts as a conduit allowing two processes to communicate.



Pipes

- Four issues:
  - Is the 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?

- Two common pipes
  - Ordinary pipes and named pipes

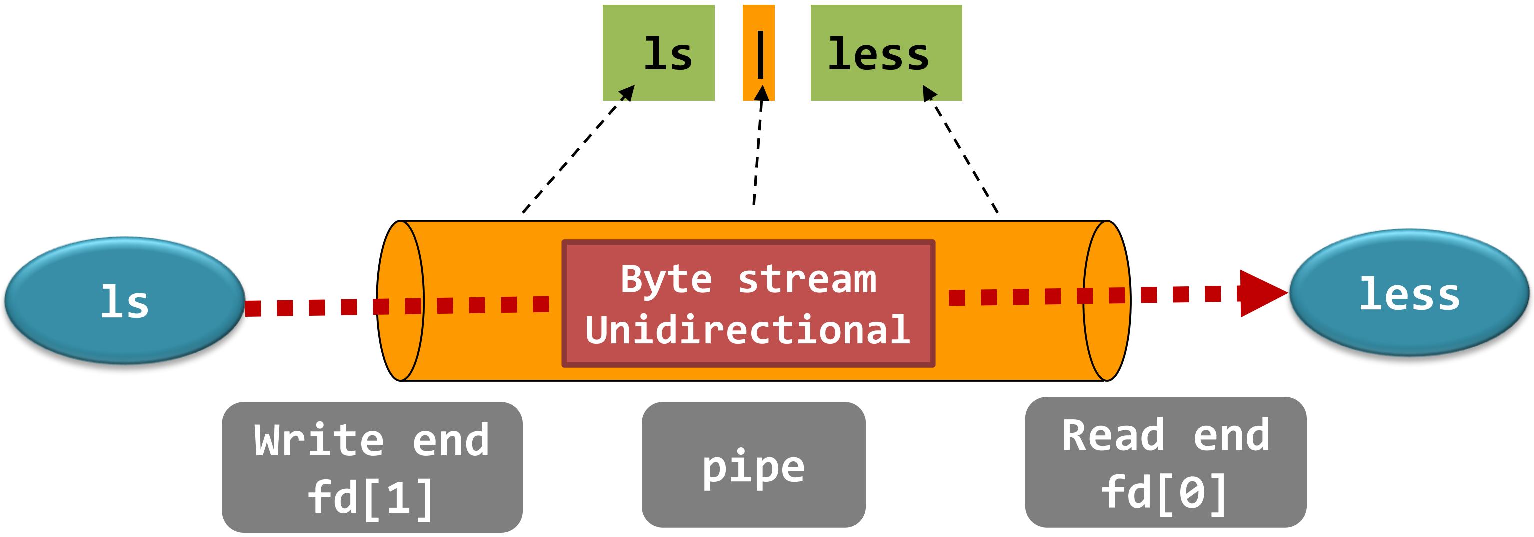
Ordinary Pipes

- Ordinary pipes (no name in file system)
  - Ordinary pipes are used only for related processes (parent-child relationship)
    - Processes must reside on the same machine
  - Ordinary pipes are unidirectional (one-way communication)
  - Ceases to exist after communication has finished

- Ordinary pipes allow communication in standard producer-consumer style
  - Producer writes to one end (**write-end**)
  - Consumer reads from the other end (**read-end**)

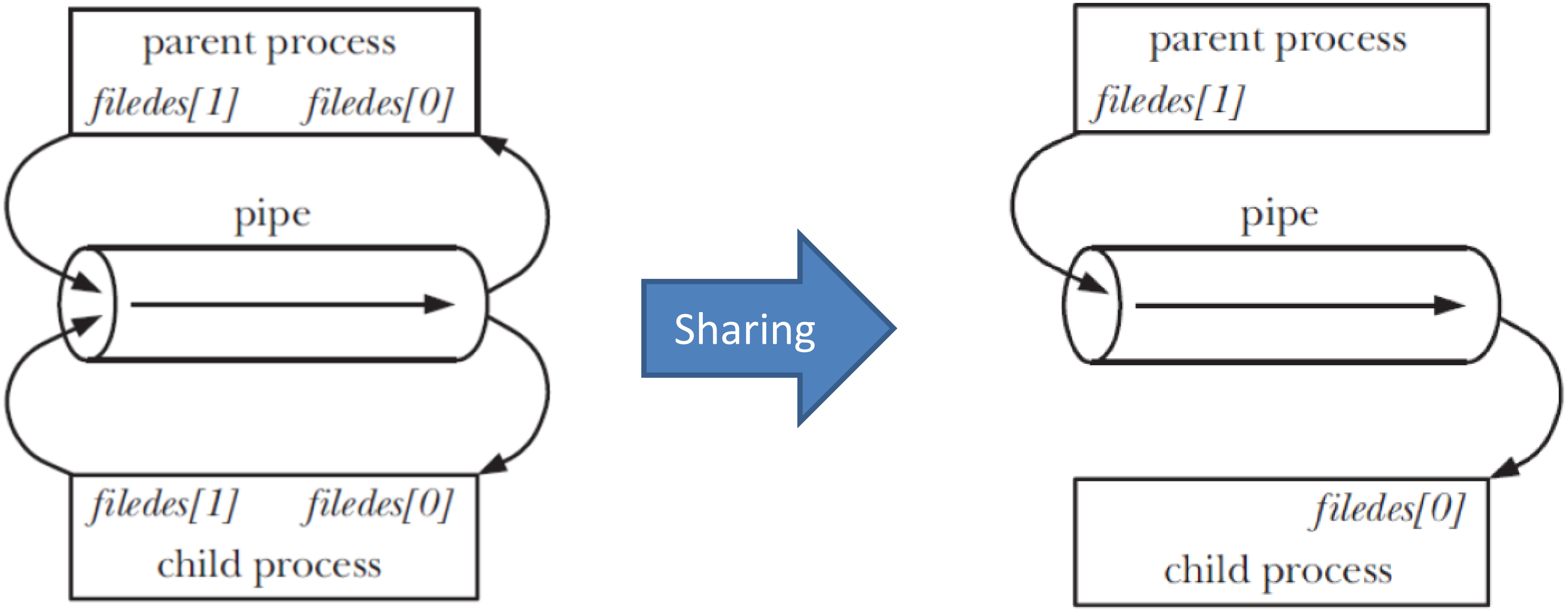
UNIX Pipe

- UNIX treats a pipe as a special file (child inherits it from parent)
  - Create: **`pipe(int fd[]);`**
    - **`fd[0]`**: read end
    - **`fd[1]`**: write end
  - Access: Ordinary **`read()`** and **`write()`** system calls



UNIX Pipe

- Pipes are anonymous (no name in file system), then how to share?
  - **`fork()`** duplicates parent's file descriptors
  - Parent and child use each end of the pipe



UNIX Pipe

```
/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* 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]);
}
else { /* 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("read %s",read_msg);

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

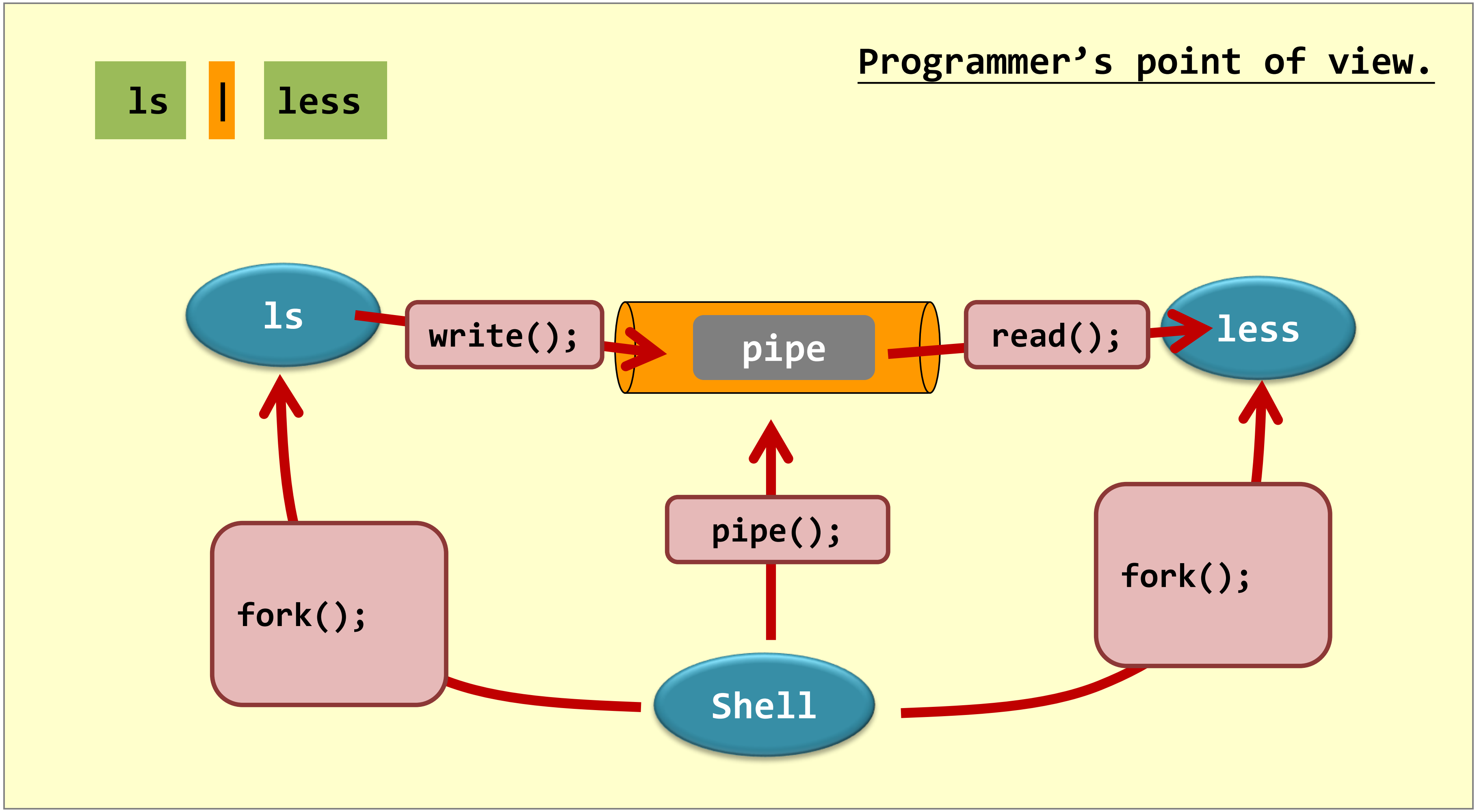
Create a child process

Parent process
Use the write end only

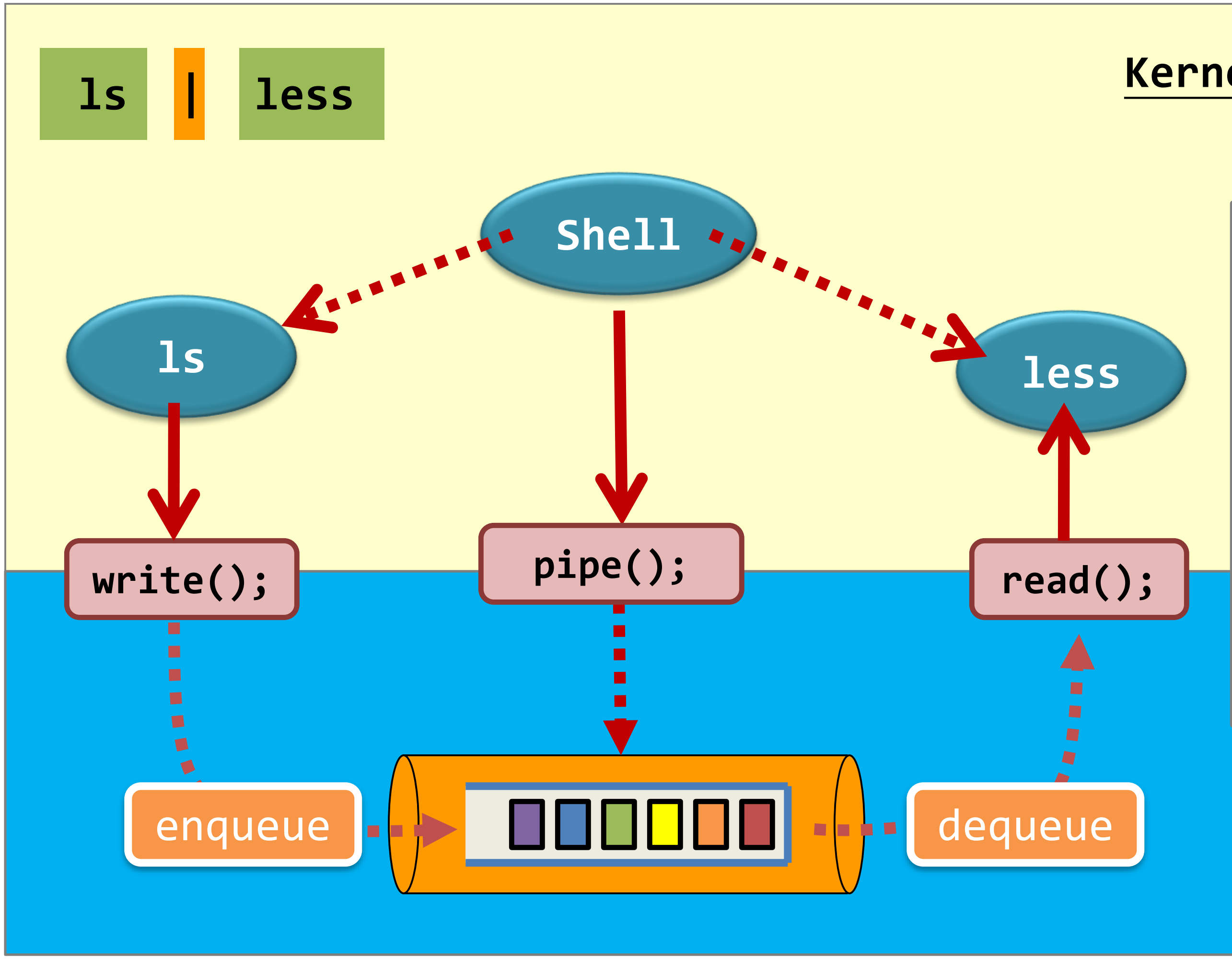
unidirectional (one-way communication

Child process
Use the read end only

Pipe - Shell Example



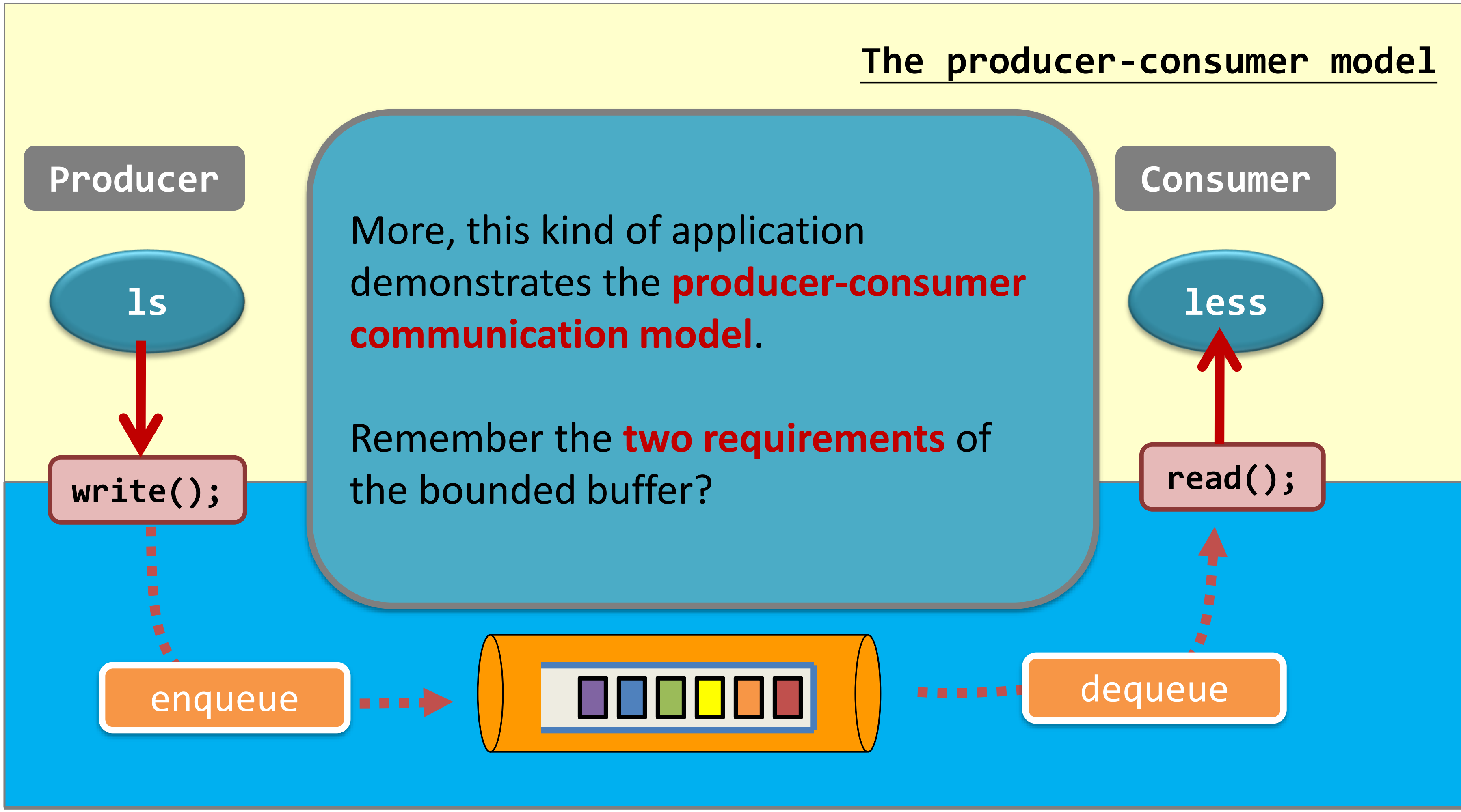
Pipe – Shell Example



The **pipe()** system call creates a piece of **shared storage in the kernel space**!

Yet, the pipe is more than a storage: **it is a FIFO queue with finite space**.

Pipe – Shell Example



More, this kind of application demonstrates the **producer-consumer communication model**.

Remember the **two requirements** of the bounded buffer?

Named Pipes

- Named pipes (pipe with name in file system)
  - No parent-child relationship is necessary (processes must reside on the same machine)
  - Several processes can use the named pipe for communication (may have several writers)
  - Continue to exist until it is explicitly deleted
  - Communication is bidirectional (still half-duplex)

- Named pipes are referred to as FIFOs in UNIX
  - Treated as typical files
  - **`mkfifo(), open(), read(), write(), close()`**

Story so far…

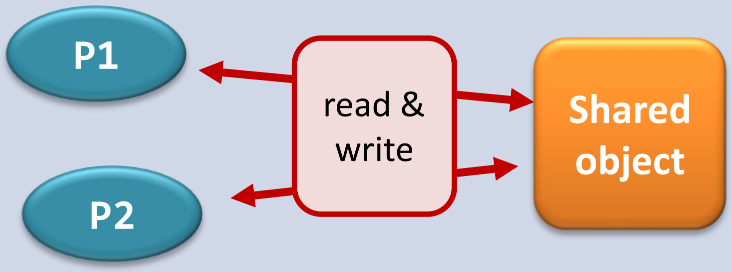
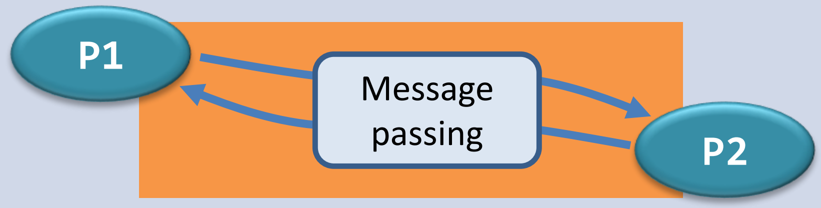
- Interprocess communication (IPC)
  - Necessary for cooperating processes
  - Producer-consumer model

- IPC models
  - Shared memory & message passing

- IPC schemes
  - Shared memory
  - Ordinary pipes (parent-child processes)
  - FIFOs (processes on the same machine)
  - Sockets (intermachine communication)

- More: Michael Kerrisk, "The Linux Programming Interface" (http://www.man7.org/tlpi/)

IPC models – another point of view

Shared Objects	Message Passing
	
<u>Challenge.</u> Coordination can only be done by detecting the status of the shared object. *E.g., is the pipe empty / full?*	**<u>Challenge.</u>** Coordination relies on the reliability and the efficiency of the communication medium (and protocol).
E.g., pipes, shared memory, and regular files.	E.g., socket programming, message passing interface (MPI) library.