Chapter 3: Processes (Part 3)

Chapter 3: Processes

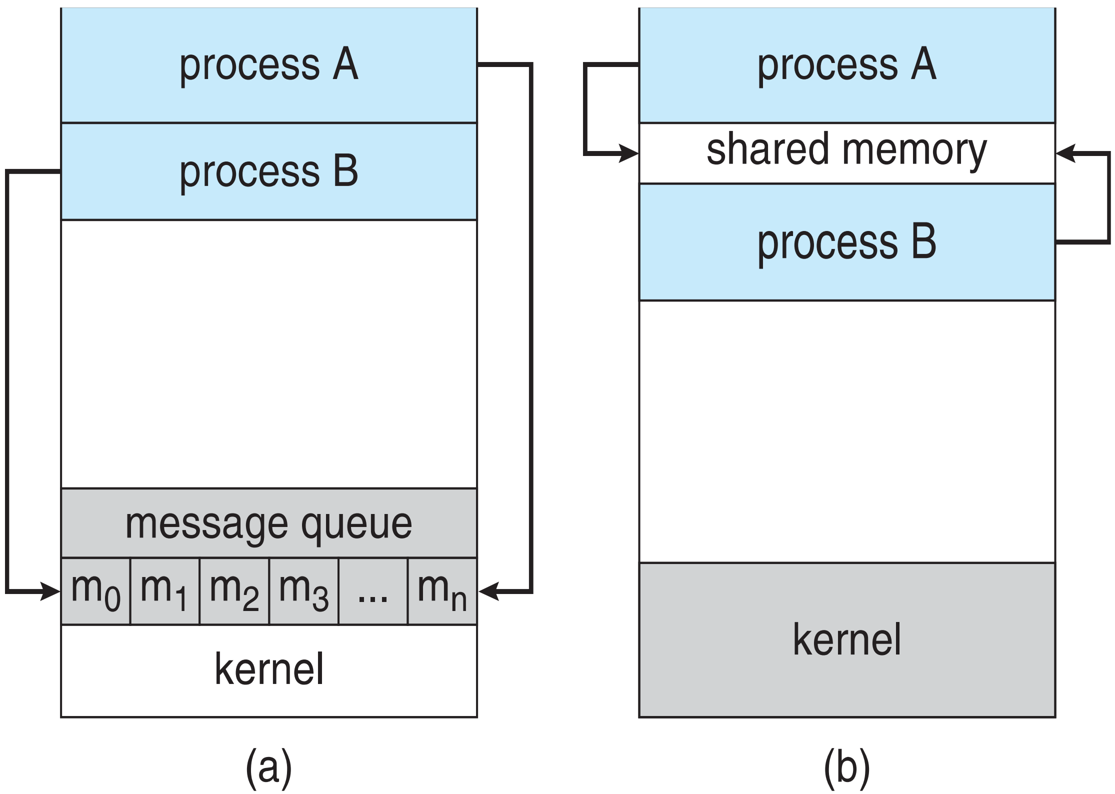
- Process Concept
- Process Scheduling
- Operations on Processes
- **Interprocess Communication (Part 3)**

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing; many users sharing the same file
  - Computation speedup; in multi-core systems
  - Modularity; recall chapter 2
  - Convenience; same user working on many tasks at the same time
- Examples:
  - Processes use and update shared data such as shared variables, memory, files, and databases.

Communications Models

- Cooperating processes need interprocess communication (IPC) mechanism to exchange data and information
- Two models of IPC
  - Many OS's implement both IPC models
- Shared memory; easier to implement and faster
- Message passing; useful for exchanging small amounts of data

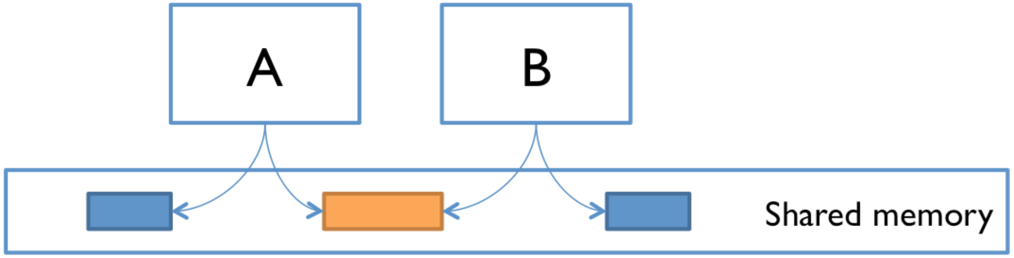


(a) Message passing. (b) shared memory.

Message Passing vs Shared Memory

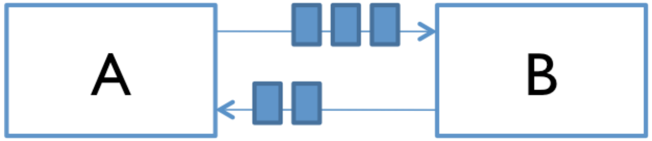
Shared Memory

- **Why good?** Performance. Set up shared memory once, then access w/o crossing protection domains
- **Why bad?** Things change behind your back → error prone



Message passing

- **Why good?** Can be used on multiple computers
- **Why bad?** Overhead. Data copying, cross protection domains



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
  - useful for exchanging small amounts of data
    - Processes can agree to remove this restriction in shared-memory systems
- The communication is under the control of the users processes not the operating system.
  - Application programmer explicitly writes the code for sharing memory
  - Processes ensure that they not write to the same location simultaneously
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  - Solution to the producer-consumer problem
    - Discussed in following slides
- Synchronization is discussed in great details in ***Chapter 5***.

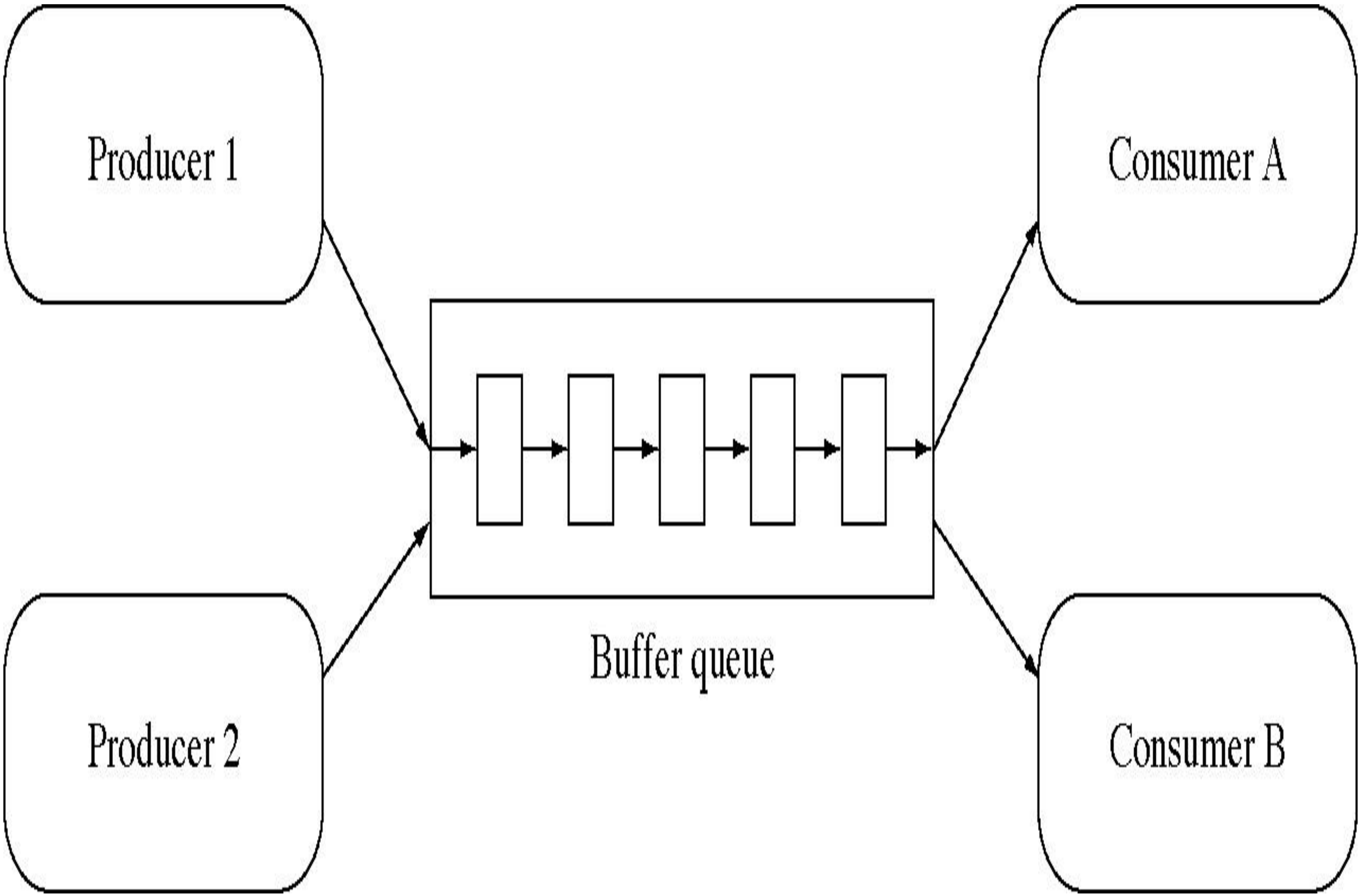
Shared-Memory Systems

- Producer-Consumer Problem
  - Paradigm for cooperating processes, ***producer*** process produces information that is consumed by a ***consumer*** process
  - Example:
    - Producer: Compiler outputs (produces) an assembly code
    - Consumer: Assembler assembles (consumes) the assembly code
    - Provides a metaphor for the client-server paradigm
      - Server = producer. Ex: web server provides HTML files/images
      - Client = consumer. Ex: client web browser reads HTML files/images
- **Solution:** producer and consumer processes share a buffer (**shared-memory**)

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Multiple Producers and Consumers

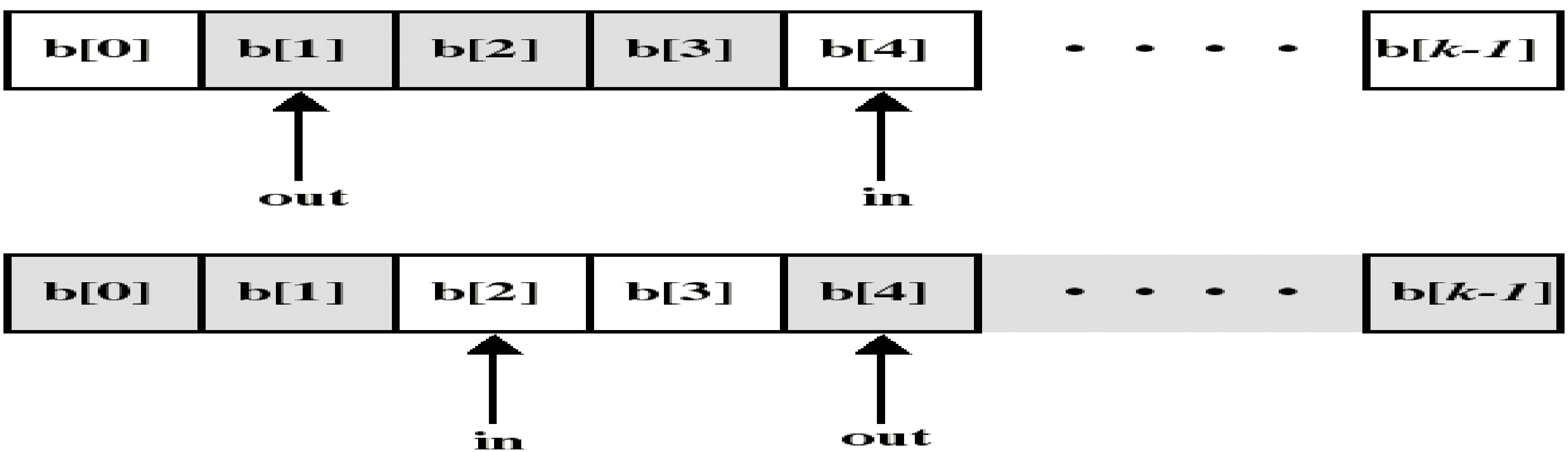


Producer/Consumer Dynamics

- A producer process produces information that is consumed by a consumer process.
- At any time, a producer activity may create some data.
- At any time, a consumer activity may want to accept some data.
- The data should be saved in a buffer until they are needed.
- If the buffer is finite, we want a *producer* to ***block*** if its new data would overflow the buffer.
- We also want a *consumer* to ***block*** if there are no data available when it wants them.
- **Summary:**
  - Synchronization: ***consumer*** should not consume data not yet produced
  - ***Producer*** should not write into a full buffer
  - All data written by the producer must be read exactly once by the consumer

Idea for Producer/Consumer Solution

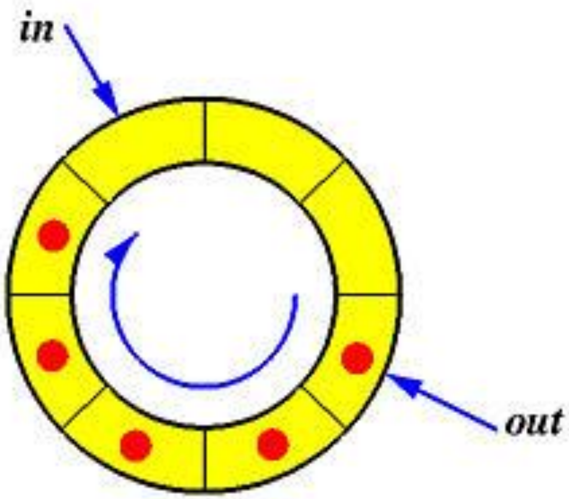
- The bounded buffer is implemented as a ***circular array*** with 2 logical pointers: **in** and **out**.
- The variable `in` points to the **next free** position in the buffer.
- The variable `out` points to the **first full** position in the buffer.



Bounded-Buffer – Shared-Memory Solution

- **Shared data:**

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

   item buffer[BUFFER_SIZE];
   int in = 0;   //'in' is the next empty slot to be filled by producer
   int out = 0;   //'out' is the next filled slot to be read by consumer
```



How to advance the index of the circular buffer?

- The ***in*** and ***out*** pointer variable in a circular array advance in the following manner:

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, ... and so on...

  because the **indices** will ***wrap*** **around**.

- How to increment index in a circular array:
  i = (i + 1) % n
  where n = the number of indices
- Example: A consumer after reading buffer will increment out as:

```
out  = (out + 1) % 4   // will increase read as:
                       //   0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, ...
```

How to check if buffer is empty/full before consumer/produce operation?

- Consumer Process:
  - How to check if buffer is empty?

```
item next_consumed;
while (true) {
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in next consumed */
}
```

- Producer Process:
  - How to check if buffer is full?

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

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

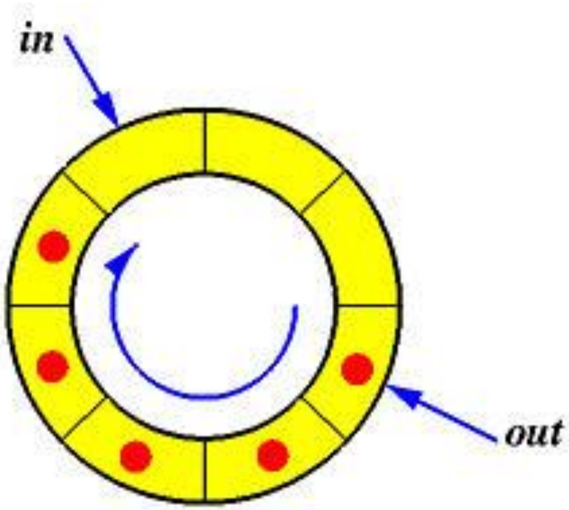
How to check if buffer is empty/full before consumer/produce operation?

- **Consumer Process:**
  - How to check if buffer is empty?
  - The buffer is empty if:
    out = = in

- **Producer Process:**
  - How to check if buffer is full?
  - The buffer is full if:
    out == ((in + 1) % BUFFER_SIZE)

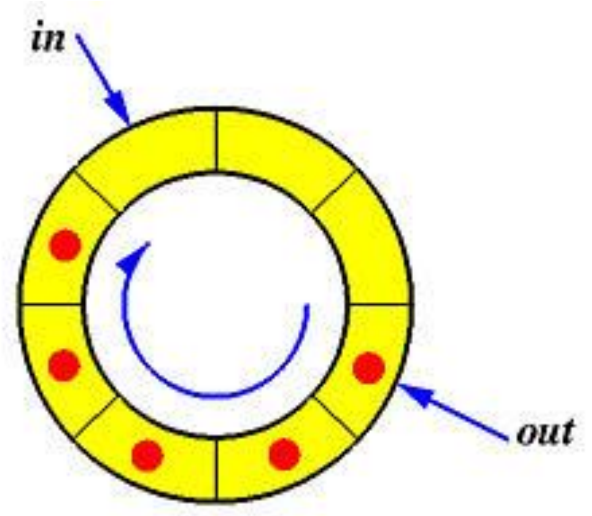
Bounded-Buffer – Producer

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



Bounded Buffer – Consumer

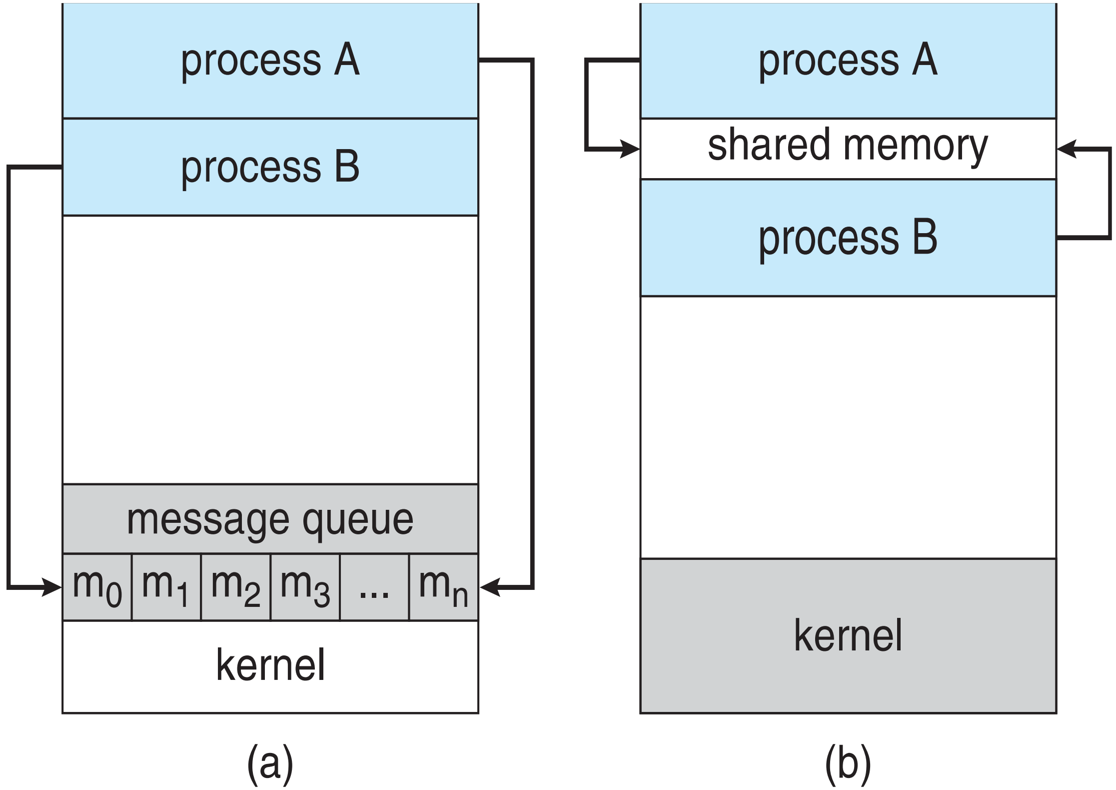
```
item next_consumed;

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

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



- Now the Solution is correct.
- The solution allows at most n – 1 items in buffer (of size n) at the same time.
  - Find out if we can use all the slots in the buffer?

Communications Models

- Cooperating processes need interprocess communication (IPC) mechanism to exchange data and information
- Two models of IPC
    - Many OS's implement both IPC models
- Shared memory; easier to implement and faster
- Message passing; useful for exchanging small amounts of data



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

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
  - Useful when communicating processes are in different computers
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - A communication link must exist between communicating processes, then
  - Communication 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: (we are concerned only with its logical implementation)
  - 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.)

- Methods of ***logically*** implementing a link:
    - (we are concerned only with its logical implementation)
  - Physical:
    - Shared Memory
    - Hardware Bus
    - Network
  - Logical
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

MP: 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*
- With this scheme:
  - Exactly one link exists between each pair of communicating processes.
  - These links may be established for processes that need to communicate before they run.
- 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

MP: Direct Communication

- Direct communication schemes
  - **Symmetric:** both sender and receiver must ***name*** the other to communicate
  - **Asymmetric:** only the sender ***names*** the recipient
    - send (P, message) – send a message to process P
    - receive(message) – receive a message from any process id

Symmetric Naming

Process P Sender	Process Q Receiver
. . `send(Q, message);` . .	. . `receive(P, &message);` . .

Asymmetric Naming

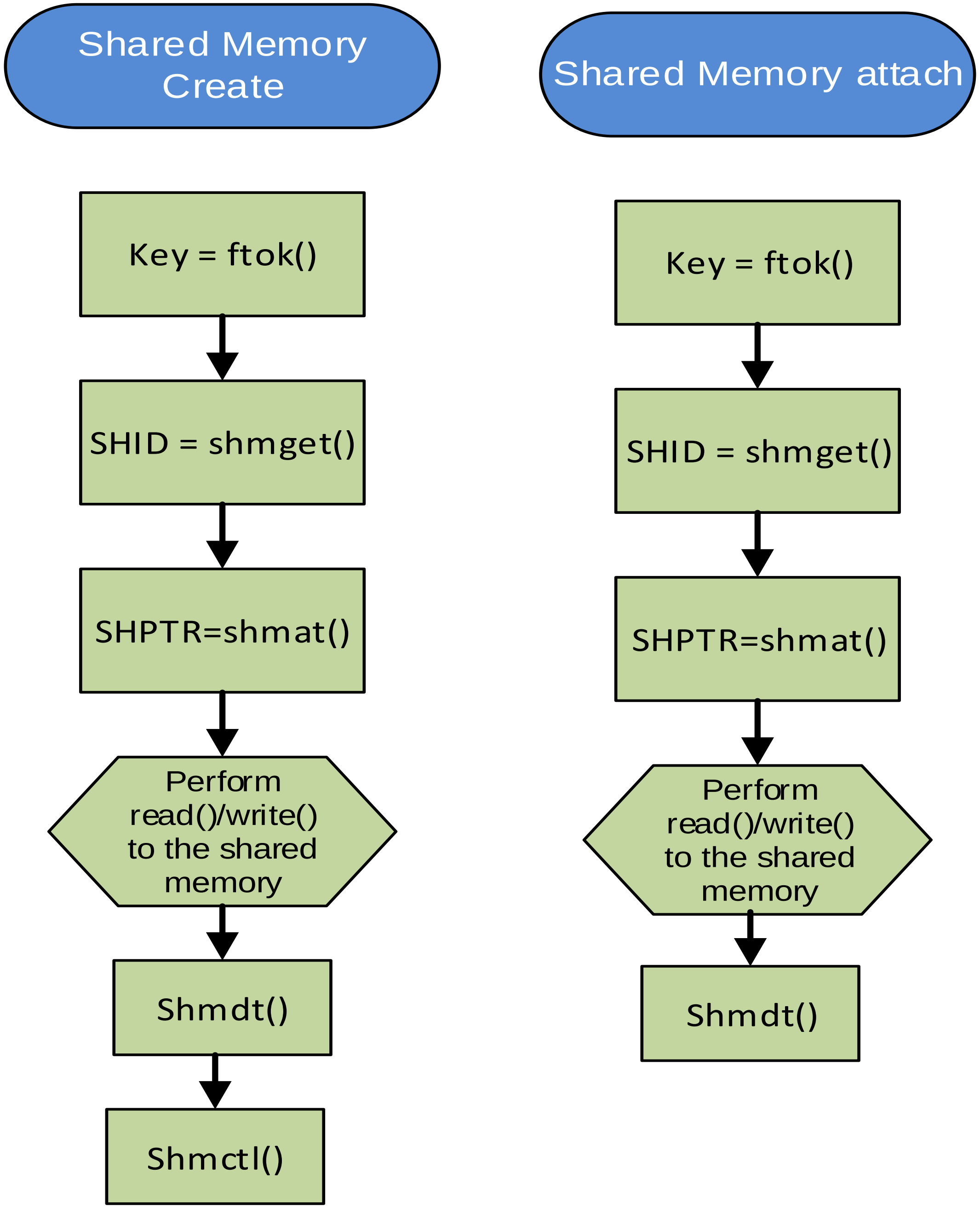
Process P Sender	Process Q Receiver
. . `send(Q, message);` . .	. . `receive(&message);` . .

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- **Examples of IPC Systems**

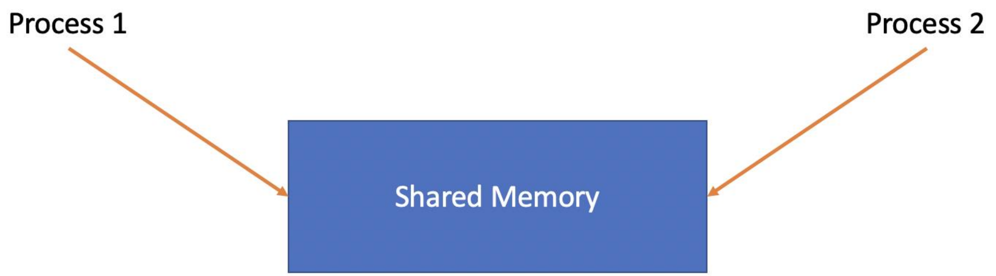
IPC Example: Steps in creating and using Shared Memory

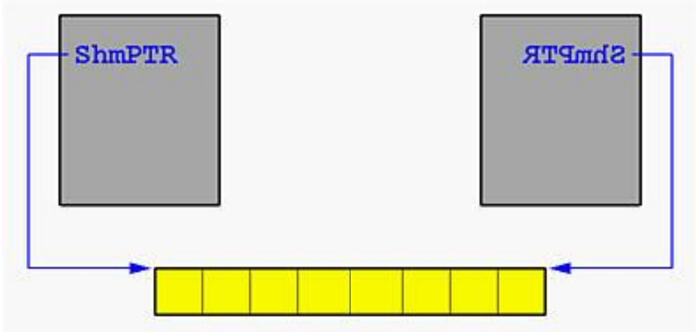
- Two cooperating processing communicating with each other using shared-memory.
- One Process creates memory segments and share it with other process.
- Detail on next slides.



IPC Example: Shared Memory

- Following steps are needed to establish shared memory between processes.
- **Step 1:** Find a key using **ftok()** function. Key is of type integer that is used to identifying shared memory segments.
- **Step 2:** Use **shmget()** to allocate a shared memory using key obtained in **Step 1**.
- **Step 3:** Use **shmat()** to attach a shared memory to an address space using ID obtained in Step 2.
  - Now using *ShmPTR* both producer and consumer can read/write ti the shared memory.
- **Step 4:** Finally use **shmdt()** to detach a shared memory from an address space and deallocate the shared memory using **shmctl()** .





Producer Process (Shared Memory)

```c
#include <sys/ipc.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

    struct Information {
    int  a;
    int b;
    char c;
    };

void main(int argc, char *argv[]) {

    key_t           ShmKEY;
    int         ShmID;
    struct Information  *ShmPTR;
    //Step 1:
    ShmKEY = ftok("myfile", 'a');
    //Step 2:
    ShmID = shmget(ShmKEY, sizeof(struct Information), IPC_CREAT | 0666);
    //Step 3:
    ShmPTR = (struct Information *) shmat(ShmID, NULL, 0);
    ShmPTR->a = 5;  ShmPTR->b = 15;     ShmPTR->c = 'z';

    //Step 4:
    shmdt((void *) ShmPTR);

    while(1);
    shmctl(ShmID, IPC_RMID, NULL);
    exit(0);
}
```

Producer writes in shared memory

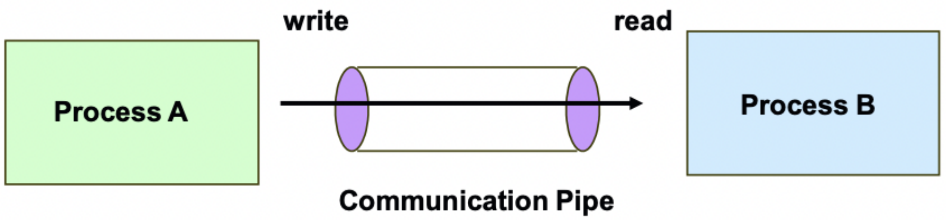
Consumer Process (Shared Memory)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

struct Information {
    int  a;
    int b;
    char c;
    };
void main(void) {

    key_t           ShmKEY;
    int             ShmID;
    struct Information  *ShmPTR;
//Step 1:
    ShmKEY=ftok("myfile", 'a');
//Step 2:
    ShmID = shmget(ShmKEY, sizeof(struct Information), 0666);
//Step 3:
    ShmPTR = (struct Information *) shmat(ShmID, NULL, 0);

    printf("%d %d %c\n", ShmPTR->a, ShmPTR->b, ShmPTR->c);

//Step 4:
    shmdt((void *) ShmPTR);
    exit(0);
}
```

Consumer reads from shared memory

IPC Example : Pipes

- Acts as a conduit allowing two processes 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?
- **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.

Use of ordinary pipes

- Two processes, ***ls*** and ***more*** use pipe to communicate with each other. One is producing information and the other is consuming information.

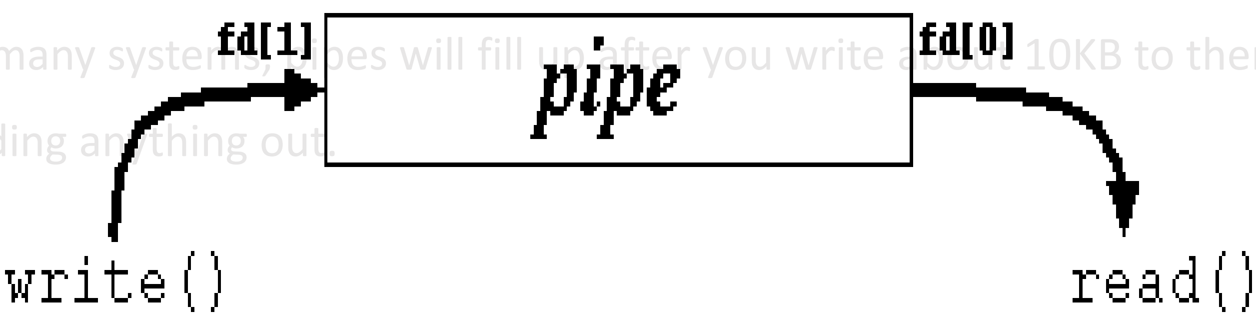
PIPES IN PRACTICE

Pipes are used quite often in the UNIX command-line environment for situations in which the output of one command serves as input to another. For example, the UNIX `ls` command produces a directory listing. For especially long directory listings, the output may scroll through several screens. The command `more` manages output by displaying only one screen of output at a time; the user must press the space bar to move from one screen to the next. Setting up a pipe between the `ls` and `more` commands (which are running as individual processes) allows the output of `ls` to be delivered as the input to `more`, enabling the user to display a large directory listing a screen at a time. A pipe can be constructed on the command line using the | character. The complete command is

```
ls | more
```

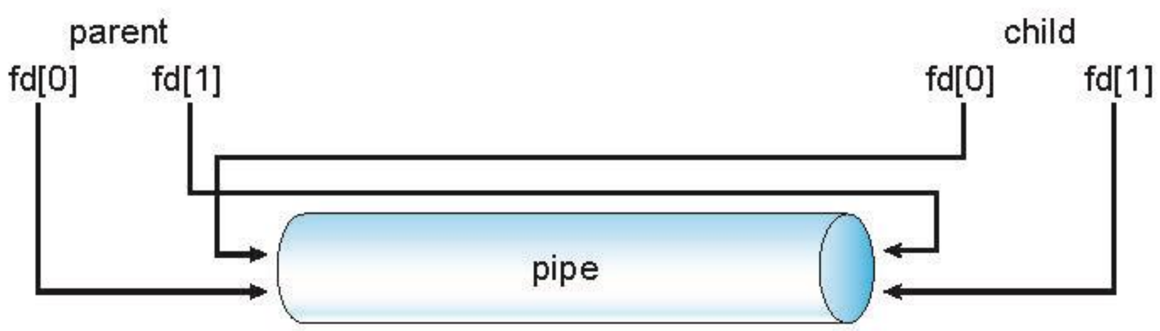
Pipes

- There is no form of IPC that is simpler than pipes.
  - A direct communication in which unidirectional channels are established between “related” processes.
  - Basically, a call to the ***`int pipe(int fd[2])`*** function returns a pair of file descriptors. → fd[0] and fd[1], used for reading and writing, respectively.
  - One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end.
  - On many systems, pipes will fill up after you write about 10KB to them without reading anything out.



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
- Require parent-child relationship between communicating processes



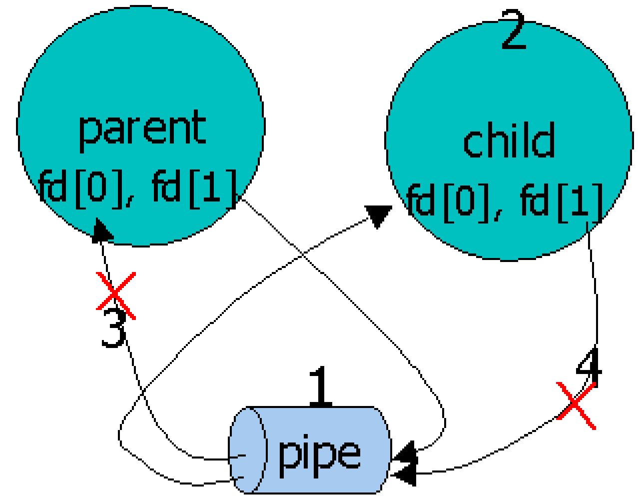
Code Example: Pipe

- System call pipe(int fds[2]) –
  - Creates a one way communication channel –
  - pipefd[2] holds the returned two file descriptors
  - Bytes written to pipefd[1] will be read from pipefd[0]

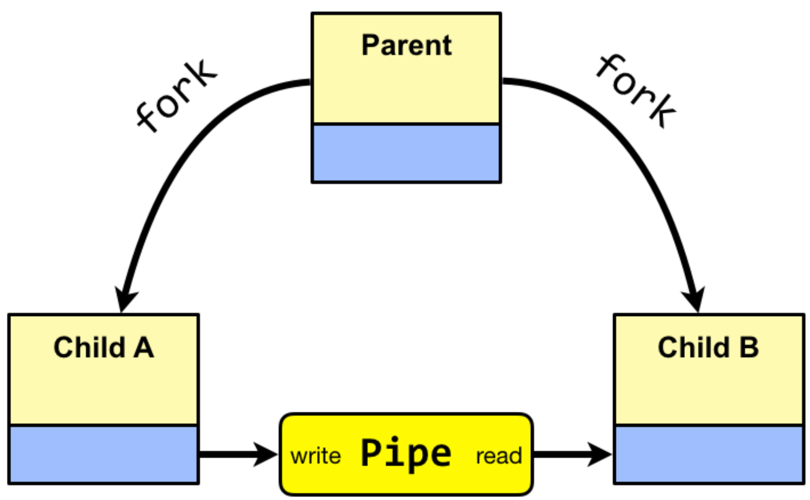
```
int pipefd[2];
pipe(pipefd); //pass address of the array.
pid=fork()
if (pid == 0) {
close(pipefd[0]);
//child process writes something to pipefd[1]
else
close(pipefd[1]);
  //read from pipefd[0]
}
```

Steps in using Pipe

- Following are the steps needed for the two process to communicate using a pipe.



1. Call Pipe() system call
2. Call fork() to create child
3. Close the readEnd of the parent, i.e., fd[0]
4. Close the write End of the child, i.e., fd[1]



Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  int pfds[2];
  char buf[30];
  pipe(pfds);
  if (fork()==0) {
      printf(" CHILD writing to the pipe\n");
      close(pfds[0]); //close readEnd of the file
      write(pfds[1], "test", 5);
      printf(" CHILD: exiting\n");
      exit(0);
  } else {
      printf("PARENT: reading from pipe\n");
      close(pfds[1]); //close writeEnd of the file
      read(pfds[0], buf, 5);
      printf("PARENT read: %s \n", buf);
      wait(NULL); } }
```

```
$ ./pipe1
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read Good Morning
```

Named Pipes

- Named Pipes also known as **FIFO**, are more powerful than ordinary pipes
- Communication is bidirectional
- it's like a pipe, except that it has a name!
- In this case, the name is that of a file that multiple processes can ***open*** and ***read*** and ***write*** to.
- No *parent-child relationship* is necessary between the communicating processes
- Several processes can use the named pipe for communication

Using Named Pipes

- `int mkfifo(const char* path, mode_t mode)`
  - Creates a named pipe
  - Can read or write to it like a regular file
  - Seen as a file in the filesystem
  - Support multiple producer processes
  - Support multiple consumer processes
- An example using command on shell

```
$ mkfifo myfifo
```

  - For example, suppose `'ls'` is the process running in the first terminal,

```
$ ls > myfifo
```

  - and you want to see its output in a different terminal..

```
$ cat < myfifo
```

FiFO Example: Producer

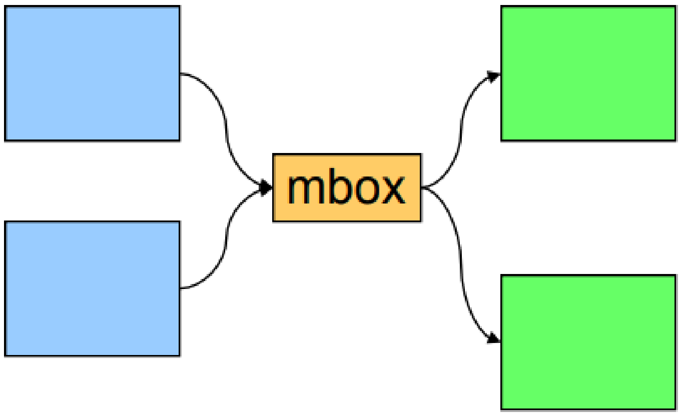
```
int main(void)
{
 char s[300];
 int num, fd;

 mkfifo(“myfifo" , 0666);
 printf("waiting for readers...\n");
 fd = open(“myfifo", O_WRONLY); //blocked
 printf("got a reader--type some stuff\n");
 write(fd, "Welcome to CS5009", sizeof("Welcome to CS5009"));
 close(fd);    /* remove the FIFO */
 unlink("myfifo);
 return 0;
}
```

FIFO Example: Consumer

```
#define MAX_BUF 1024
int main(){
int fd;

char buf[MAX_BUF];
 /* open, read, and display the message from the FIFO */
fd = open("myfifo", O_RDONLY);

read(fd, buf, MAX_BUF);
printf("Received: %s\n", buf);

close(fd);
return 0;
}
```

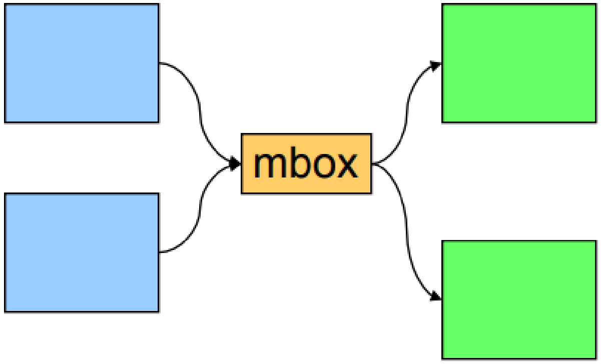
MP: Indirect Communication

- Messages are directed and received from mailboxes
  - 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



MP: Indirect Communication

- OS provides operations allowing a process to
  - create a new mailbox
    - The owner is the process that creates the mailbox M
  - Send() and receive() messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

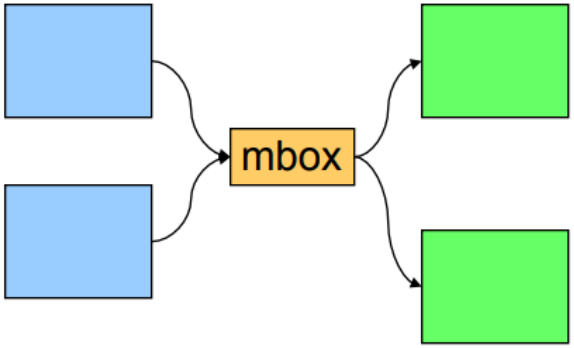


send(*A, message*) – send a message to **mailbox A**

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

MP: Indirect Communication

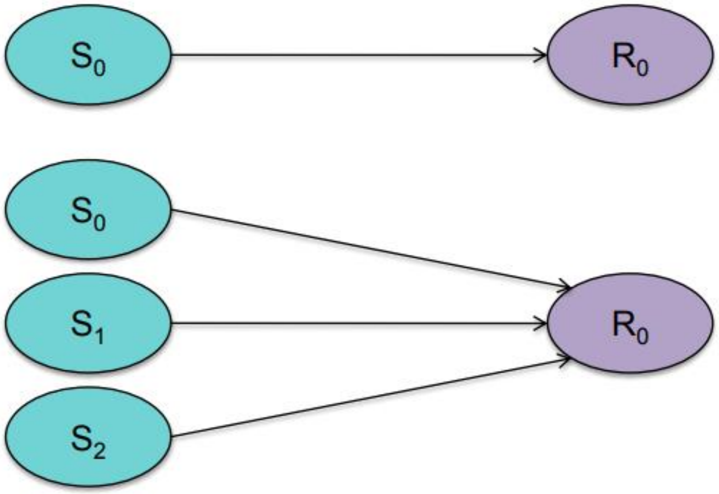
- Mailbox sharing
  - Suppose processes P1, P2, and P3 share mailbox A
  - P1, sends a message to mailbox A by executing send(A, message)
  - P2 and P3 execute receive(A, message)
    - Who gets the message? … P2 or P3 ?



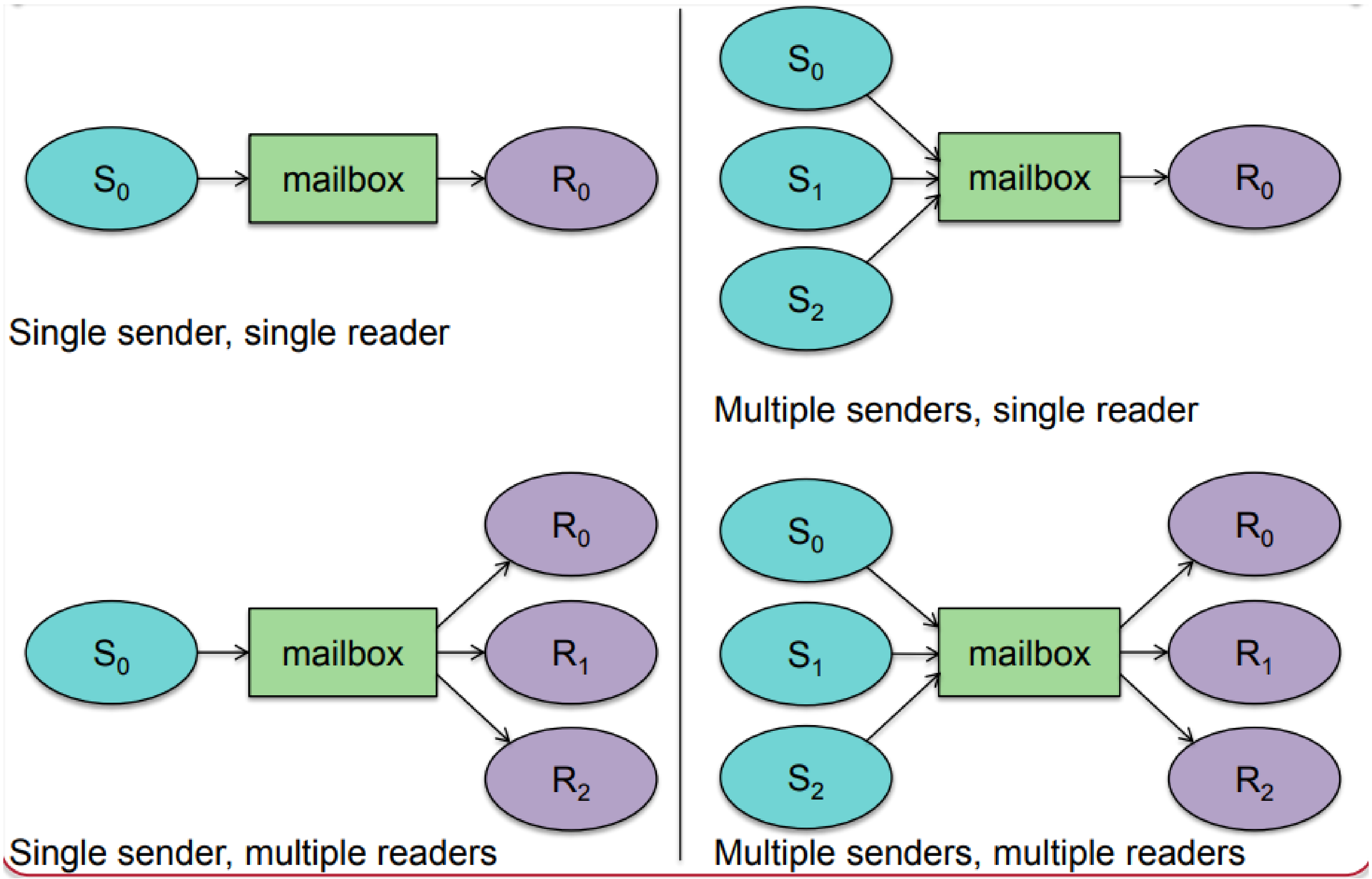
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.
    - Round robin algorithm where processes take turn in receiving messages
    - Sender is notified who the receiver was.

MP: Indirect Communication

- Sending process identifies receiving process
- Receiving process can identify sending process
  - Or can receive it as a parameter



Mailboxes



S0
mailbox
R0
Single sender, single reader
S0
S1
S2
mailbox
R0
Multiple senders, single reader
S0
mailbox
R0
R1
R2
Single sender, multiple readers
S0
S1
S2
mailbox
R0
R1
R2
Multiple senders, multiple readers

MP: 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
- A good example of **synchronous message passing** is the telephone system
  - where the caller places a call and waits for the callee to answer; the caller is blocked (and does not do more work) until the callee answers.

MP: Synchronization

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **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**
- A good example of **asynchronous message passing** is the postal system where the sender drops a piece of mail in the mailbox and continues along (shopping, eating, whatever); the recipient receives the mail sometime later.

MP: Buffering

- **The *capacity* of a link is its buffer size:**
- 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 link full

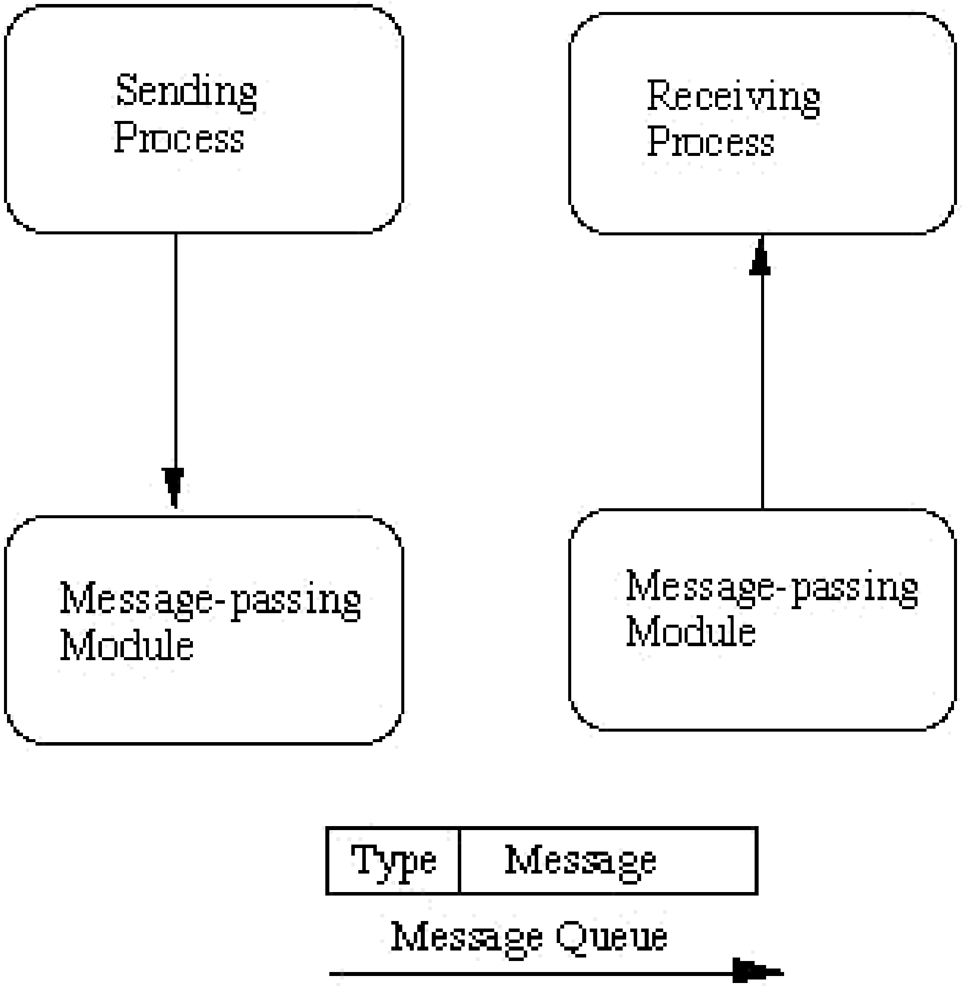
3. **Unbounded capacity** – infinite length

   Sender never blocks and the link is asynchronous.

End of Chapter 3

Example-Message Queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
  long mtype;
  char mtext[200];
};
```



Producer-Message Queue

```
int main(void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;
  key = ftok("ipc_example.c", 'B');
  msqid = msgget(key, 0644 | IPC_CREAT);
  printf("Enter lines of text, ^D to quit:\n");
  buf.mtype = 1;
 while(gets(buf.mtext), !feof(stdin)) {
    msgsnd(msqid, (struct msgbuf *)&buf,
     strlen(buf.mtext)+1, 0);
  }
  msgctl(msqid, IPC_RMID, NULL);
  return 0;
}
```

Consumer-Message Queue

```
int main(void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;

  key = ftok("ipc_example.c", 'B');
  msqid = msgget(key, 0644);
  for(;;) {
    msgrcv(msqid, (struct msgbuf *)&buf,
     sizeof(buf.mtext), 0, 0);
    printf("consumer: \"%s\"\n", buf.mtext);
  }
  return 0;
}
```