Operating System Design

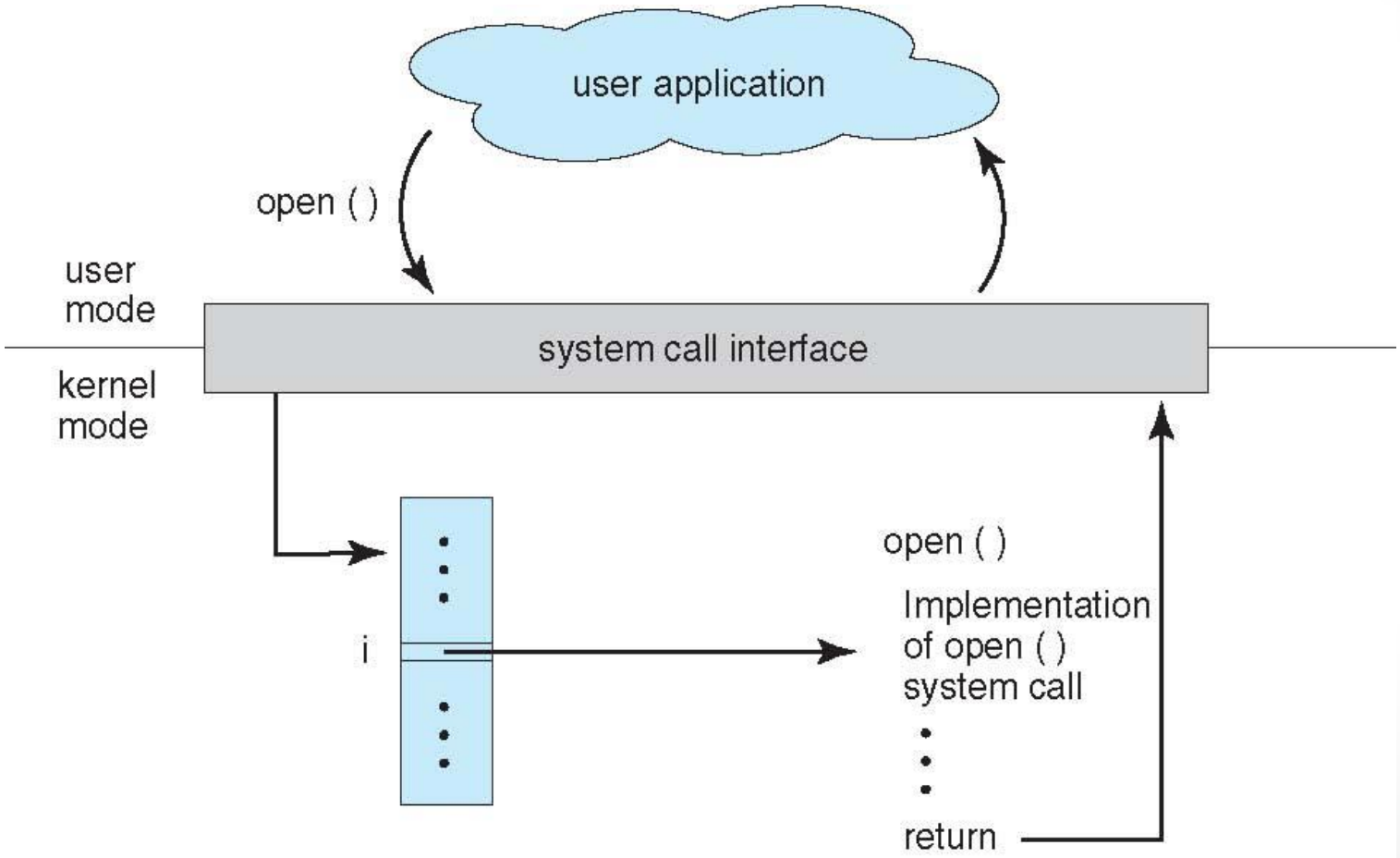
Processes Operations Review
Inter Process Communication (IPC)

Neda Nasiriani

Fall 2018

Review

- What is a system call?
  - Provides an interface to the services made available by an operating system
  - Systems execute thousands of system calls per second
    - Every file access
    - Every input/output device
- What is an API?
  - Application programming interface that specifies a set of functions available to the programmers.
- What is the Unix system's API?
  - POSIX which is accessible through C language as **libc** library
- System call interface?
  - Each programming language provides a system call interface that serves as the link to system calls made available by the operating system


user application
open ( )
user mode
system call interface
kernel mode
i
open ( )
Implementation of open ( ) system call
return

Process

Process Creation: How?

- An existing process can create a new process by calling the fork() system call
- fork() runs once in the parent process but returns two times,
  1) In the child process, with returning value of 0
  2) In the parent process, with the value of the child process id
- Both the child and the parent process start executing with the instruction that follows the fork() system call
- The child process gets a copy of the parent's data space, heap and stack (It is a separate copy from the parent's)

Process Creation using fork(): sharing resources

- The child process gets **its own copy** of the data section, stack and heap of the parent process
- The child process get a duplicate of all open file descriptors in its parent process
  - The parent and the child share a file table entry for every open descriptor
  - The parent and the child share the same file offset
    - If a child process is writing to standard output when the parent process is executing it can append to the end of standard output
  - Note that every UNIX program has three streams opened for it when it starts up, one for input (stdin), one for output (stdout), and one for error messages (stderr) with file descriptors of 0, 1 and 2 respectively.
- Does changes in the child variables change the parent variables?
  - No

Process Execution

- **A child process can execute**

  1) A segment of its parent code (as we saw before)

  2) Another program that is loaded to its memory using exec() system call

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function

Process Execution: example

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Termination

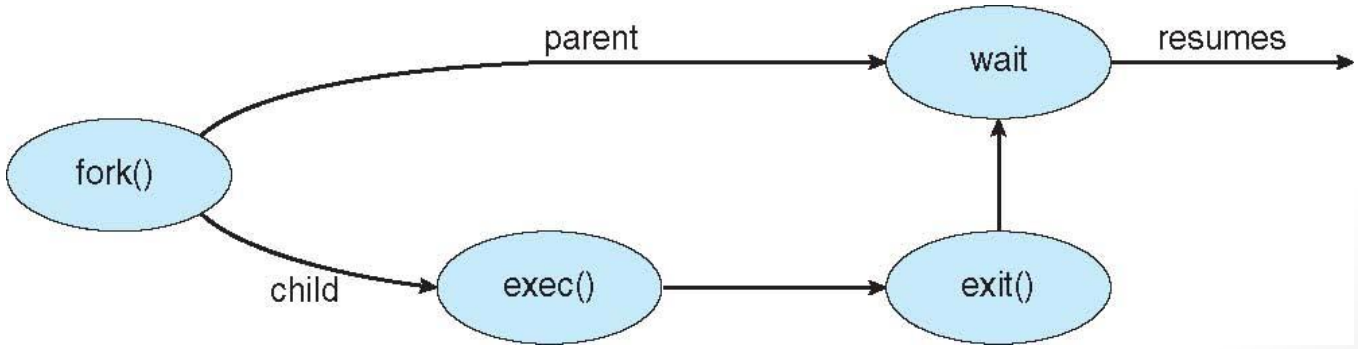
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait(&status)`**)
  - Process' resources are deallocated by operating system
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- **Zombie process?**
  - If parent has not called **`wait()`** yet but the child is terminated, the info of child is still kept in the process table. This child process is a zombie.
- **Orphan process**
  - If parent terminated without invoking **`wait`**, the child process is an orphan, which is adopted by **init** process

Process Operations Diagram

- The parent can wait on the child process by system calls
  - pid_t wait (int * status);
  - pid_t waitpid (pid_t pid, int * status, int options);
  - Both these return process id on successful return or -1 in case of an error
- Acts like a synchronization point



Demo!

ls

Interprocess Communication (IPC)

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Example: Chrome browser

Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Chrome Browser

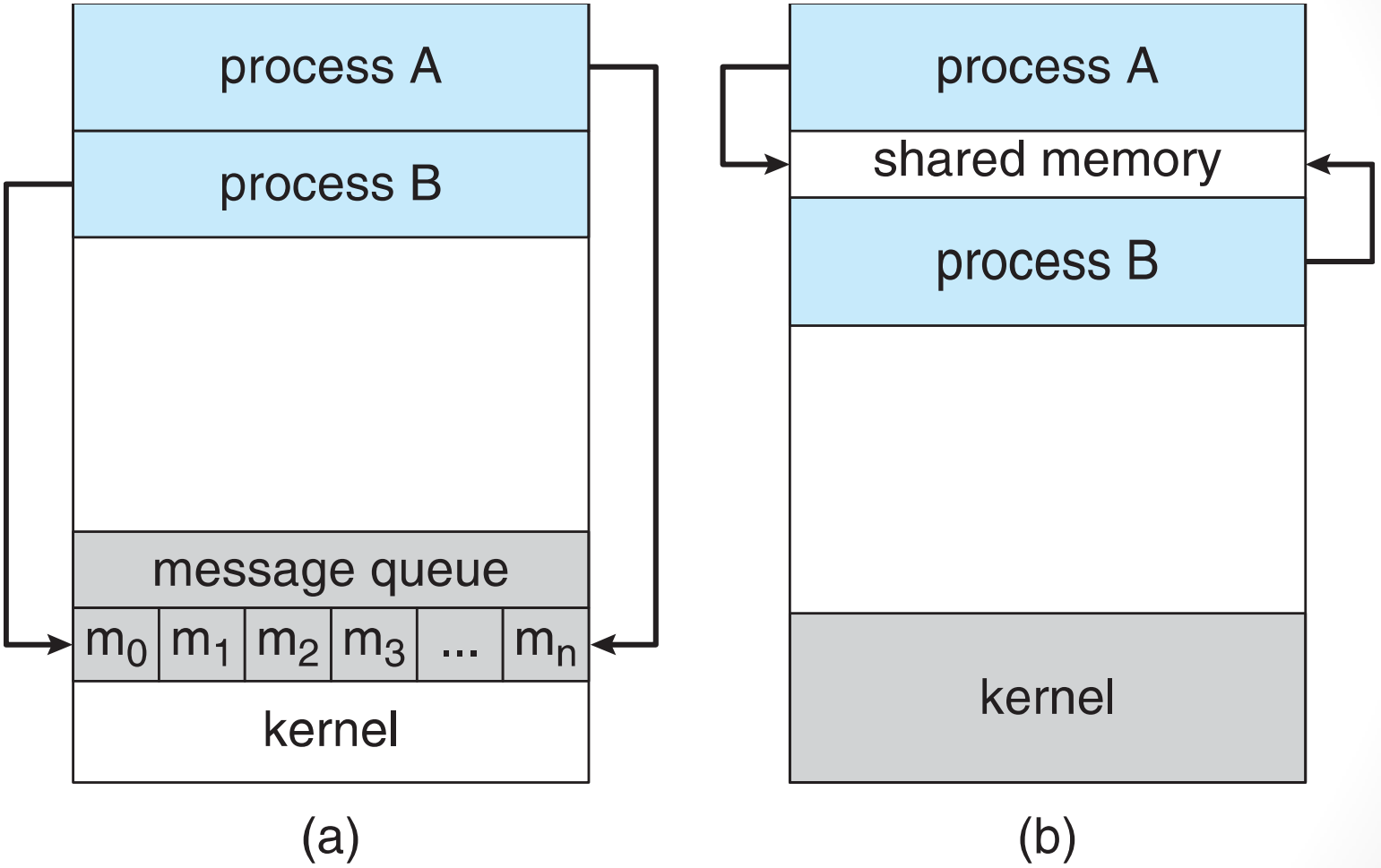
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome B… types of processes:
  - **Browser**
  - **Renderer** Javascript
    - Runs in **sa**… izing effect of security explo…
  - **Plug-in** process for each type of plug-in

Renderer needs to communicate with the browser
And plug-in process needs to talk to the Browser if a tab is using a plug-in

Interprocess Communication

- Example: Chrome browser
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communication Models



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

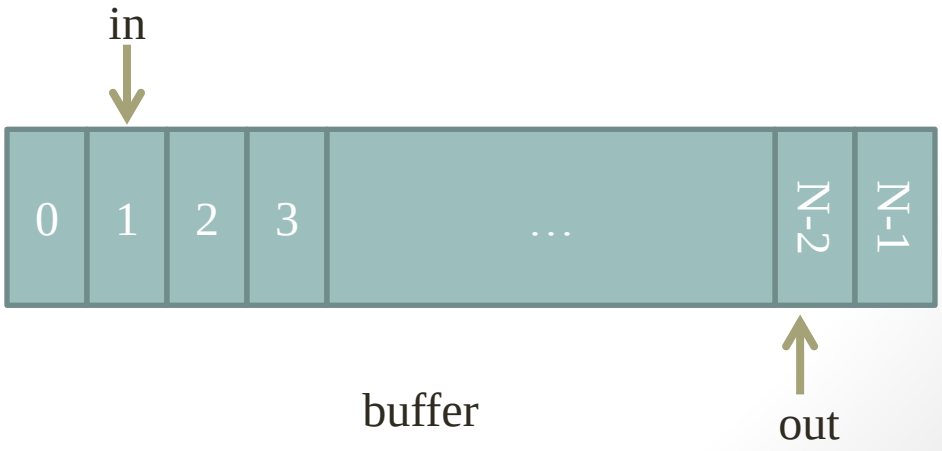
Shared Memory: Producer Consumer Example

- Assume process A is producing items and put them into a buffer and Process B is reading from the buffer
  - Example: Compiler sends assembly code to the assembler
- Buffer
  - Unbounded
  - Bounded

buffer

Shared Memory: Producer Consumer Example

- Assume process A is producing items and put them into a bounded buffer of size N and Process B is reading from the buffer
  - Example: Compiler sends assembly code to the assembler
- How can we design this communication?
  - in: next empty slot
  - out: next ready to be read slot
  - Initially in = out = 0



Shared Memory: Producer Consumer Example

- Assume process A is producing items and put them into a bounded buffer of size N and Process B is reading from the buffer
- How can we design this communication?
  - What does process A and B code look like?

B: Consumer Code:

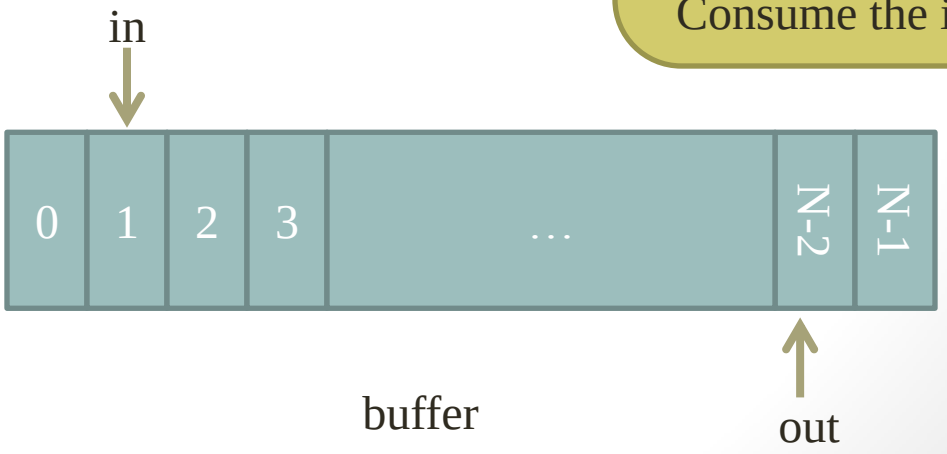
item = buffer [out]
out = (out + 1)%N

Consume the item

A:Producer Code:

Produce item

buffer [in] = item
in = (in + 1)%N



Producer Consumer Example

- What are the things that can go wrong here?
  - The buffer could be empty (nothing to be read by the consumer)
    - in == out
  - The buffer could be full (no items can be added by the producer)
    - (in + 1) % N == out
  - What if they try to access the buffer at the same time?

A:Producer Code:

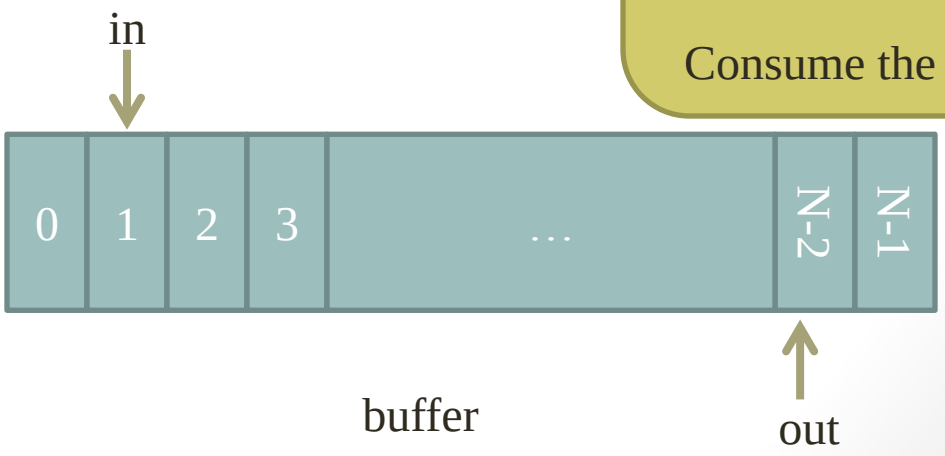
Produce item

buffer [in] = item
in = (in + 1)%N

B: Consumer Code:

item = buffer [out]
out = (out + 1)%N

Consume the item



Producer-Consumer Example

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

- What if they try to access the buffer concurrently?
- We do not allow the indexes to be the same, unless the buffer is empty.
- What is maximum usable number of slots?
- Can we improve this?
  - More on this later!!!

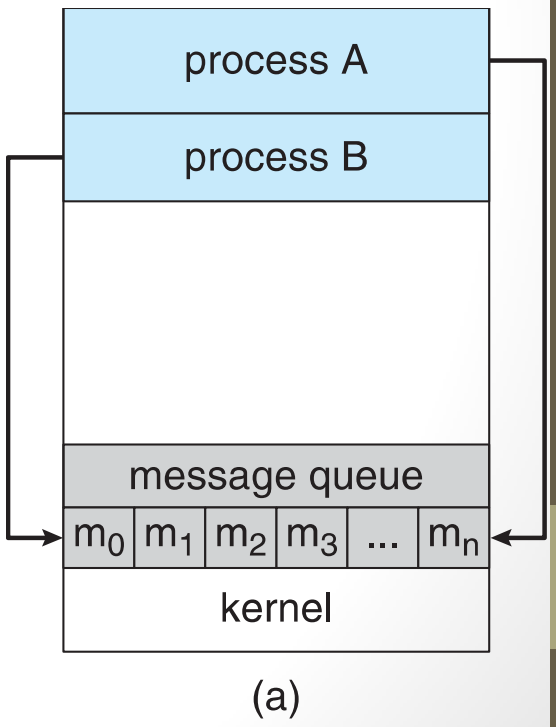
```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % N;

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

Shared Memory Conclusion

- As we saw in the producer consumer example, in message sharing the processes need to coordinate among each other to avoid loss of info and be synchronized
- This communication is fast but there are complications associated with it
- So, let's look at the other communication that is facilitated by the kernel
  - Message Passing

Message Passing

- 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



(a)

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?
  - 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 communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - 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
- 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
- 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?
- 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
- **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**

Producer Consumer: Message Passing

- 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 */
}
```

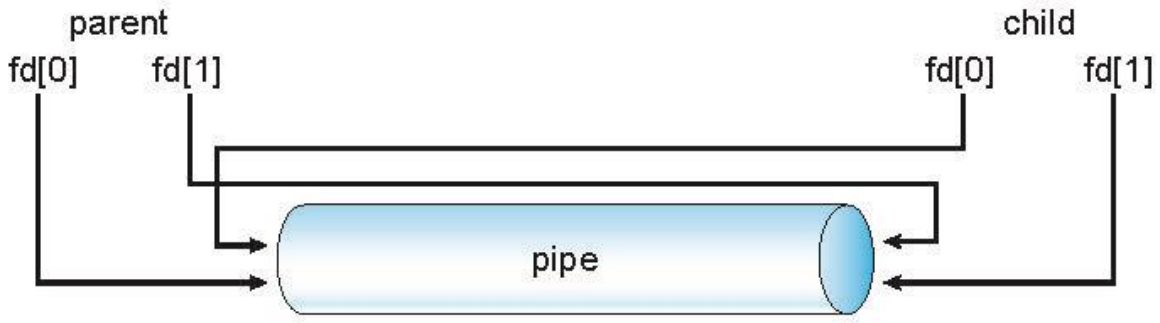
Pipes

Pipes

- Acts as a channel 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.

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



- Windows calls these **anonymous pipes**

Named Pipes

- Named Pipes 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