SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

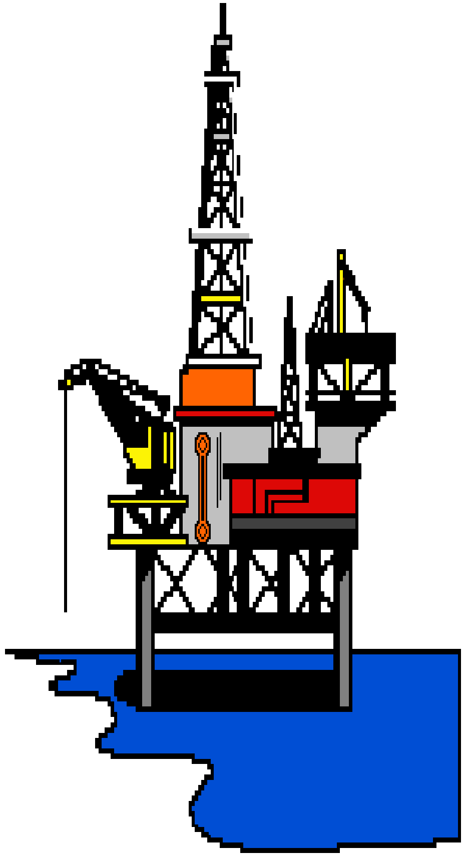
Introduction to Concurrency

Classical Problems of Concurrency

Classical Problems of Concurrency

- There are many of them – let’s briefly see three famous problems:

1. P/C Bounded-Buffer
2. Readers and Writers
3. Dining-Philosophers



Reminder: P/C problem with race condition

```
#define N 100                                  /* number of slots in the buffer */
int count = 0;                                 /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          item = produce_item( );              /* generate next item */
          if (count == N) sleep( );            /* if buffer is full, go to sleep */
          insert_item(item);                   /* put item in buffer */
          count = count + 1;                   /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);    /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
          item = remove_item( );               /* take item out of buffer */
          count = count − 1;                   /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                  /* print item */
     }
}
```

P/C Bounded-Buffer Problem

- We need 3 semaphores:

1. A semaphore **mutex** (initialized to 1) to have mutual exclusion on buffer access.
2. A semaphore **full** (initialized to 0) to synchronize producer and consumer on the number of consumable items.
3. A semaphore **empty** (initialized to n) to synchronize producer and consumer on the number of empty spaces.

Bounded-Buffer – Semaphores

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer – Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (TRUE);
```

Bounded-Buffer – Consumer Process

```
do {
    wait(full)
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (TRUE);
```

Notes on P/C Bounded-Buffer Solution

- Remarks (from consumer point of view):
  - Putting **signal(empty)** inside the CS of the consumer (instead of outside) has no effect since
    the producer must always wait for both semaphores before proceeding.
  - The consumer must perform **wait(full)** before **wait(mutex)**, otherwise deadlock occurs if consumer enters CS while the buffer is empty.
- Conclusion: using semaphores is a difficult art ... ☺

Full P/C Bounded-Buffer Solution

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

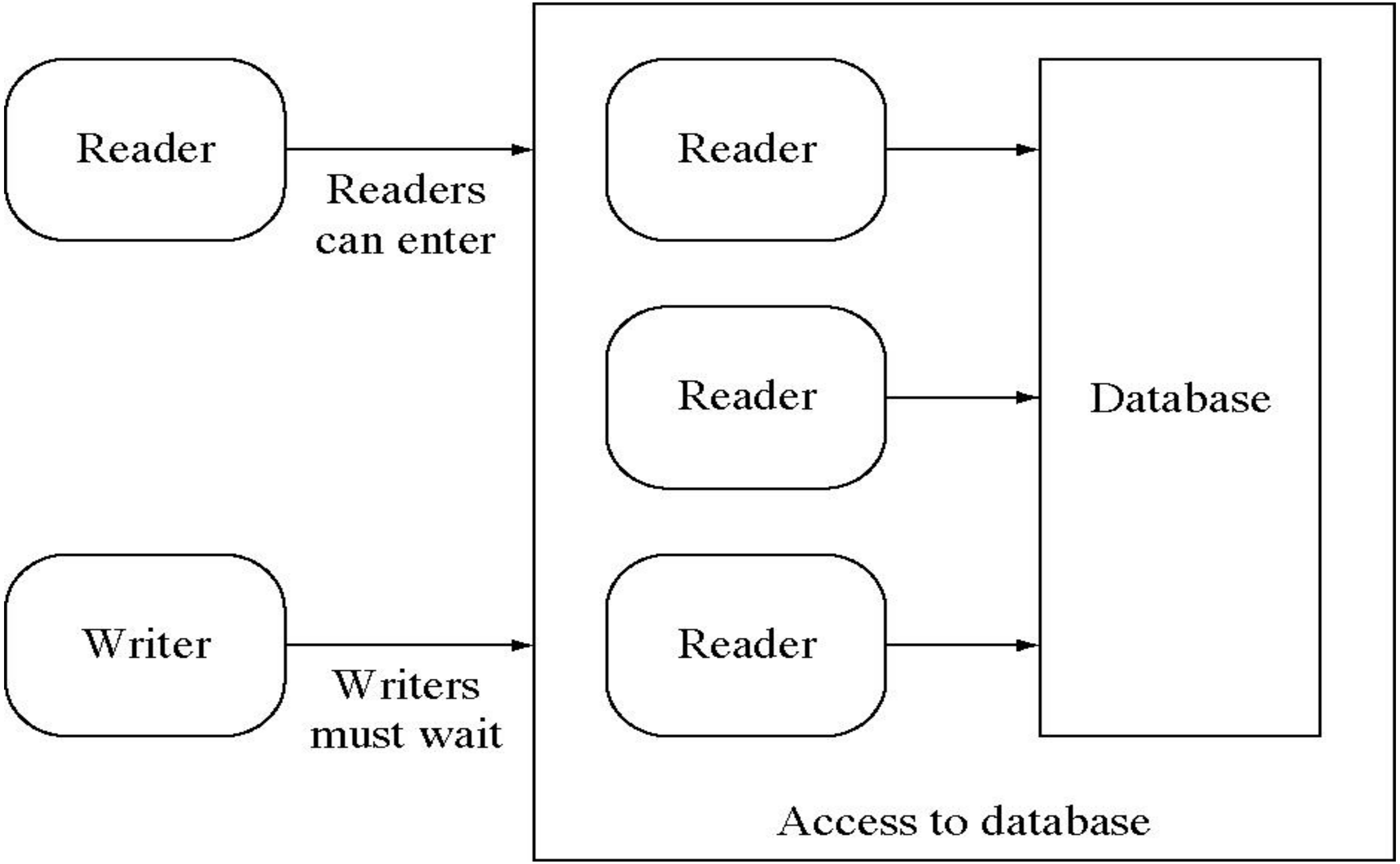
Readers-Writers Problem

- A data set/repository is shared among a number of concurrent processes:
  - Readers – only read the data set; they do **not** perform any updates.
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

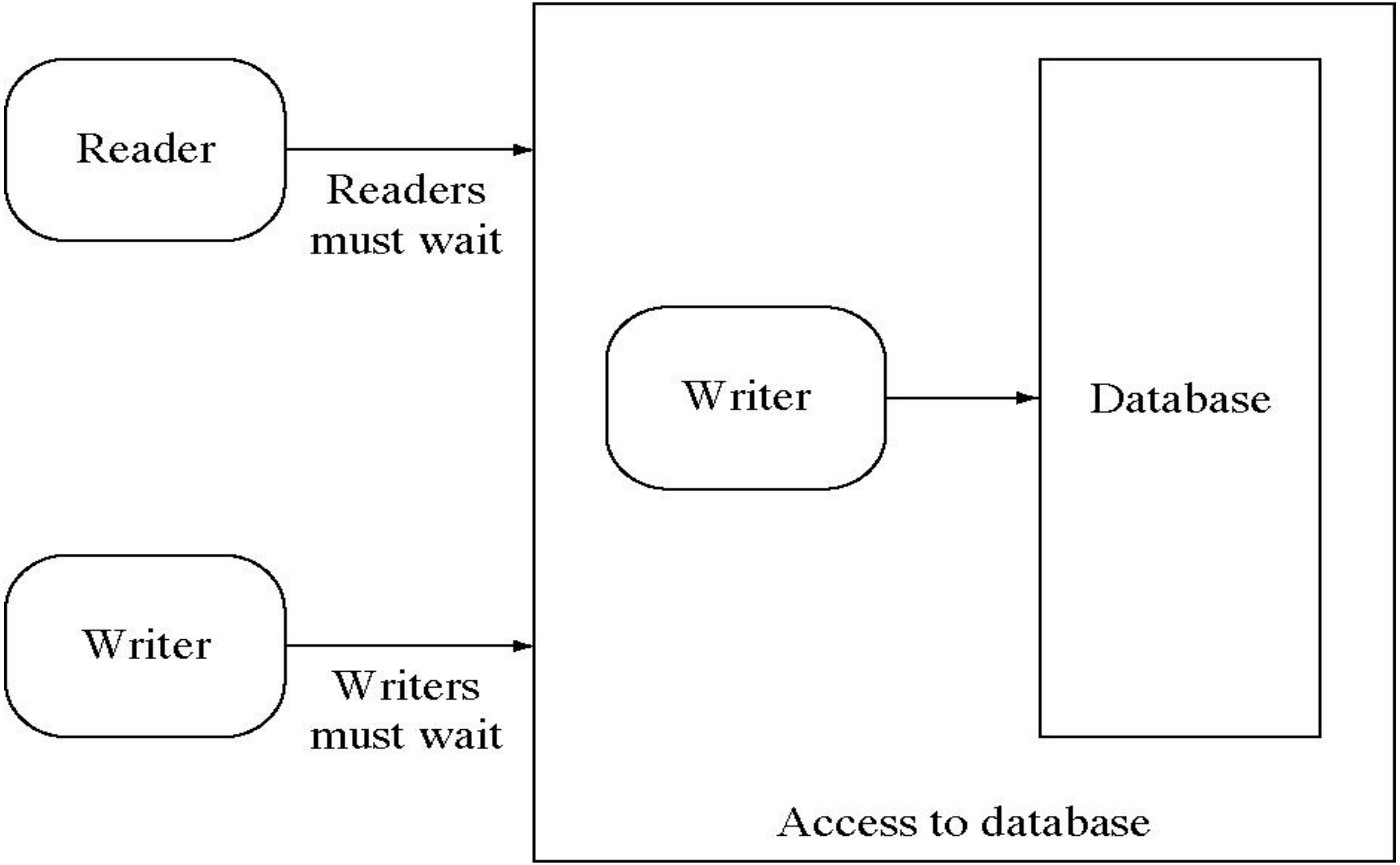
Readers-Writers Dynamics

- Any number of reader activities and writer activities are running.
- At any time, a reader activity may wish to read data.
- At any time, a writer activity may want to modify the data.
- Any number of readers may access the data simultaneously.
- During the time a writer is writing, no other reader or writer may access the shared data.

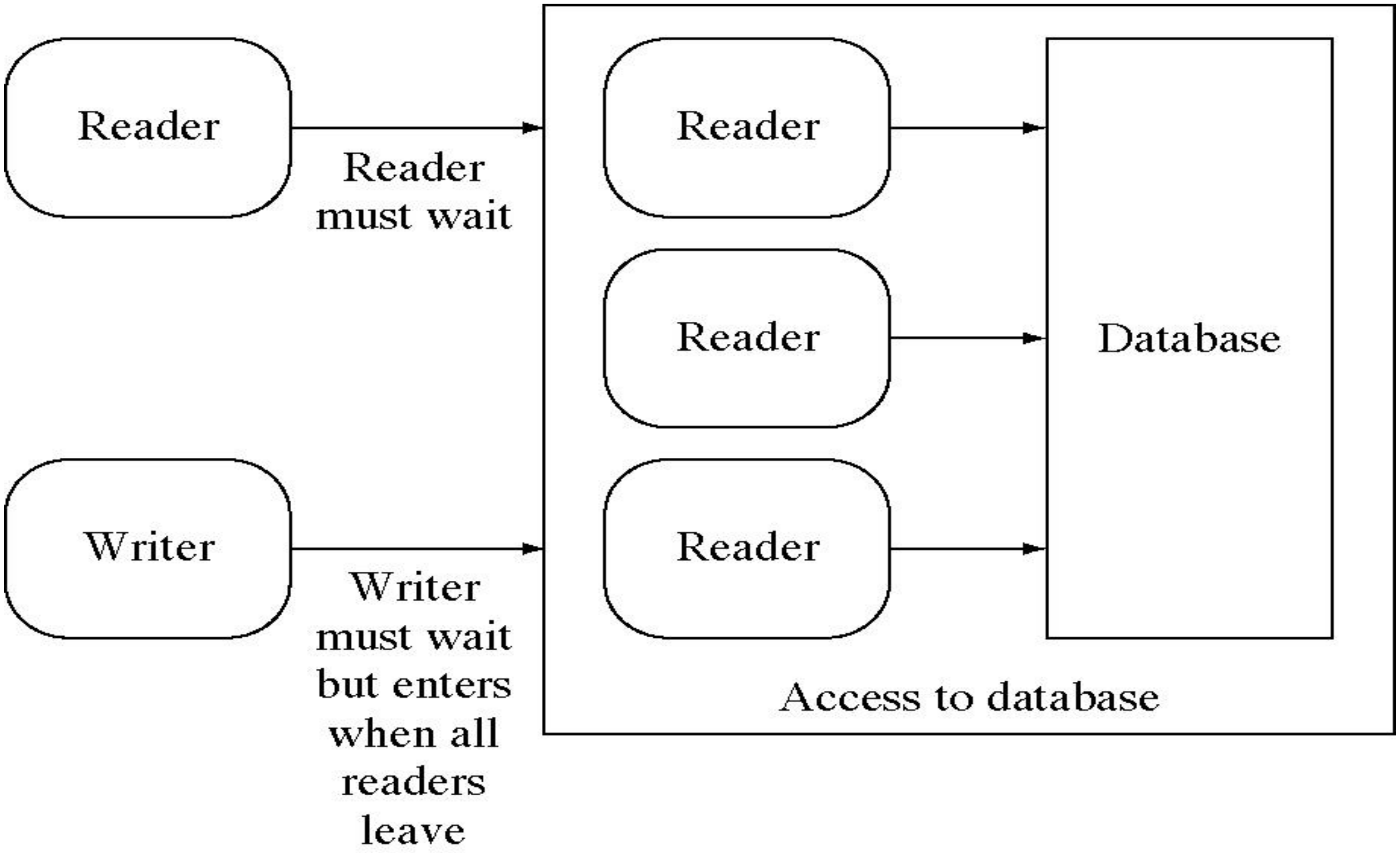
Readers-Writers with active readers



Readers-Writers with an active writer



Should readers wait for waiting writer?



Readers-Writers problem

- There are various versions with different readers and writers preferences:

1. The **first** readers-writers problem, requires that no reader will be kept waiting unless a writer has obtained access to the shared data.
2. The **second** readers-writers problem, requires that once a writer is ready, no new readers may start reading.
3. In a solution to the **first** case writers may starve; In a solution to the **second** case readers may starve.

First Readers-Writers Solution (1)

- **readcount** (initialized to 0) counter keeps track of how many processes are currently reading.
- **mutex** semaphore (initialized to 1) provides mutual exclusion for updating readcount.
- **wrt** semaphore (initialized to 1) provides mutual exclusion for the writers; it is also used by the first or last reader that enters or exits the CS.

First Readers-Writers Solution (2)

- Shared data

  semaphore mutex, wrt;
  int readcount;

  Initially

  mutex = 1, wrt = 1, readcount = 0

First Readers-Writers – Writer Process

do {

wait(wrt);

…

writing is performed

…

signal(wrt);

} while(TRUE);

First Readers-Writers – Reader Process

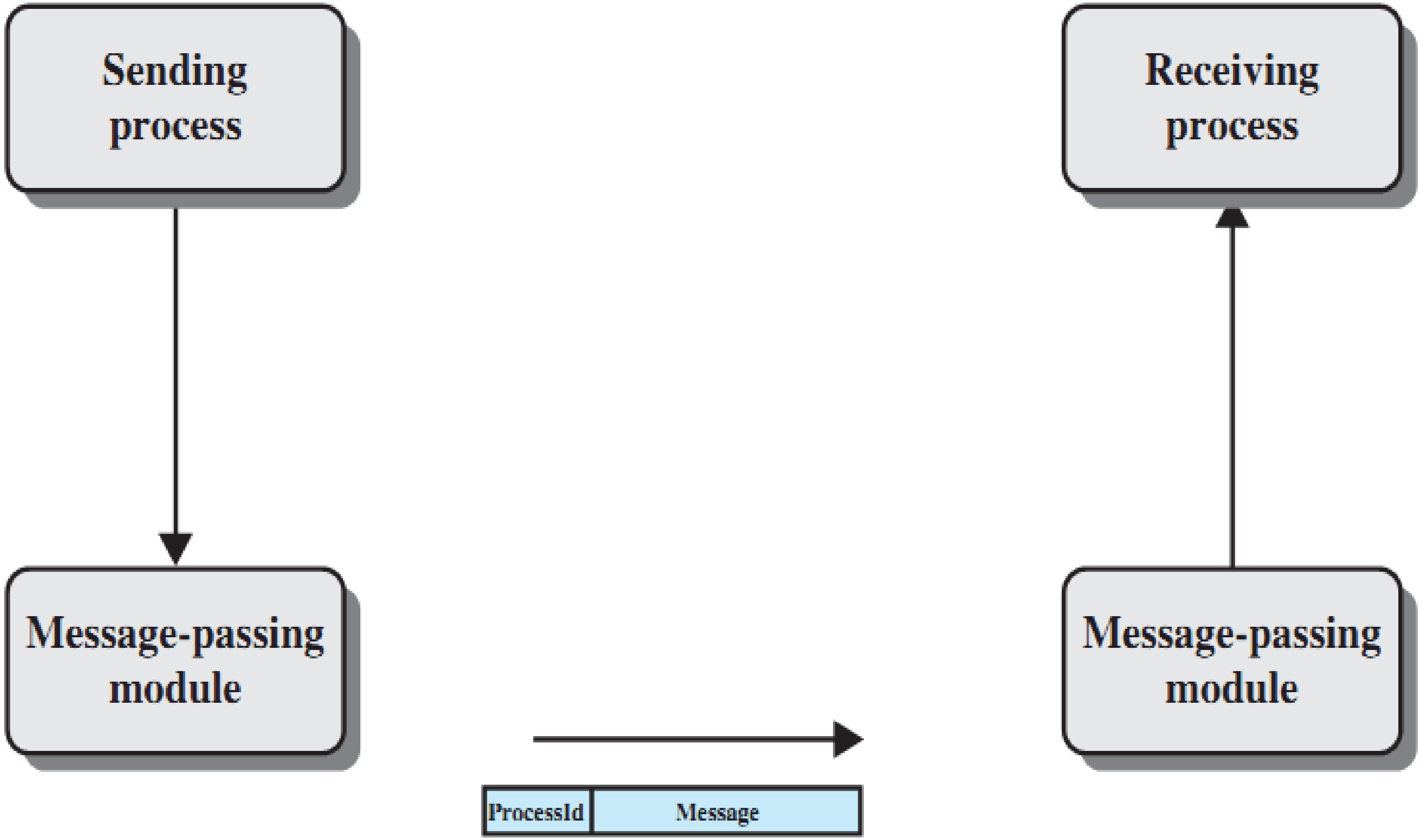
```
do {
    wait(mutex);
      readcount++;
      if (readcount == 1)
           wait(wrt);
      signal(mutex);
           ...
        reading is performed
           ...
      wait(mutex);
      readcount--;
      if (readcount == 0)
        signal(wrt);
      signal(mutex);
} while(TRUE);
```

Inter-process Communication

Inter-Process Communication (IPC)

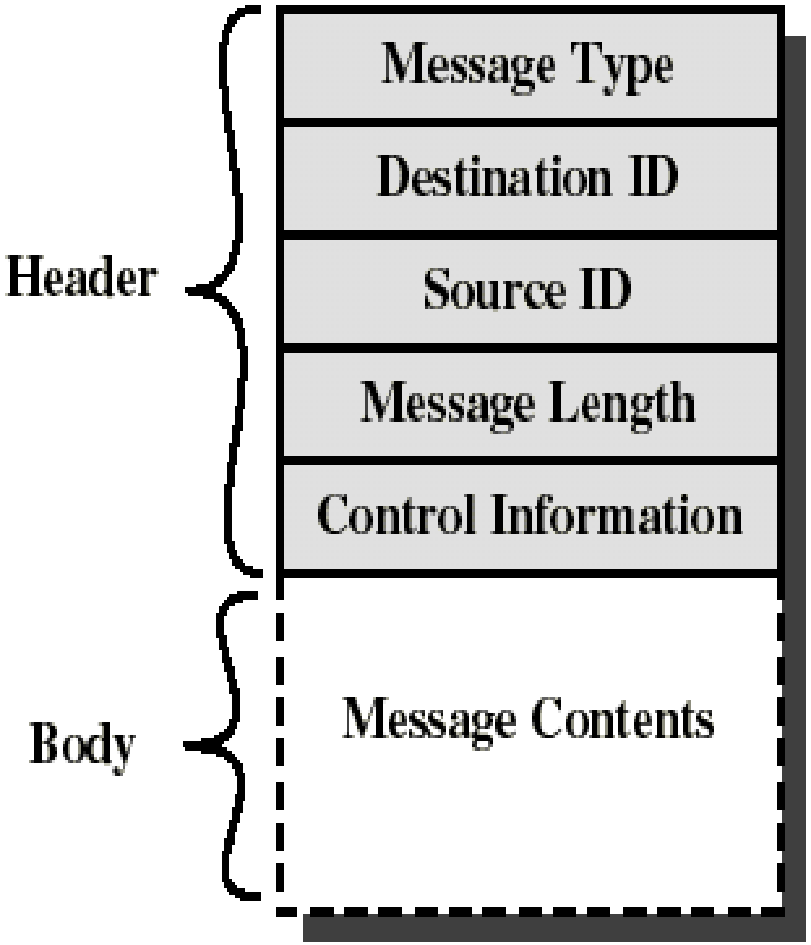
- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- We have at least two primitives:
  - **send**(*destination, message*) or **send**(*message*)
  - **receive**(*source, message*) or **receive**(*message*)
- Message size is fixed or variable.

Basic Message-passing Primitives

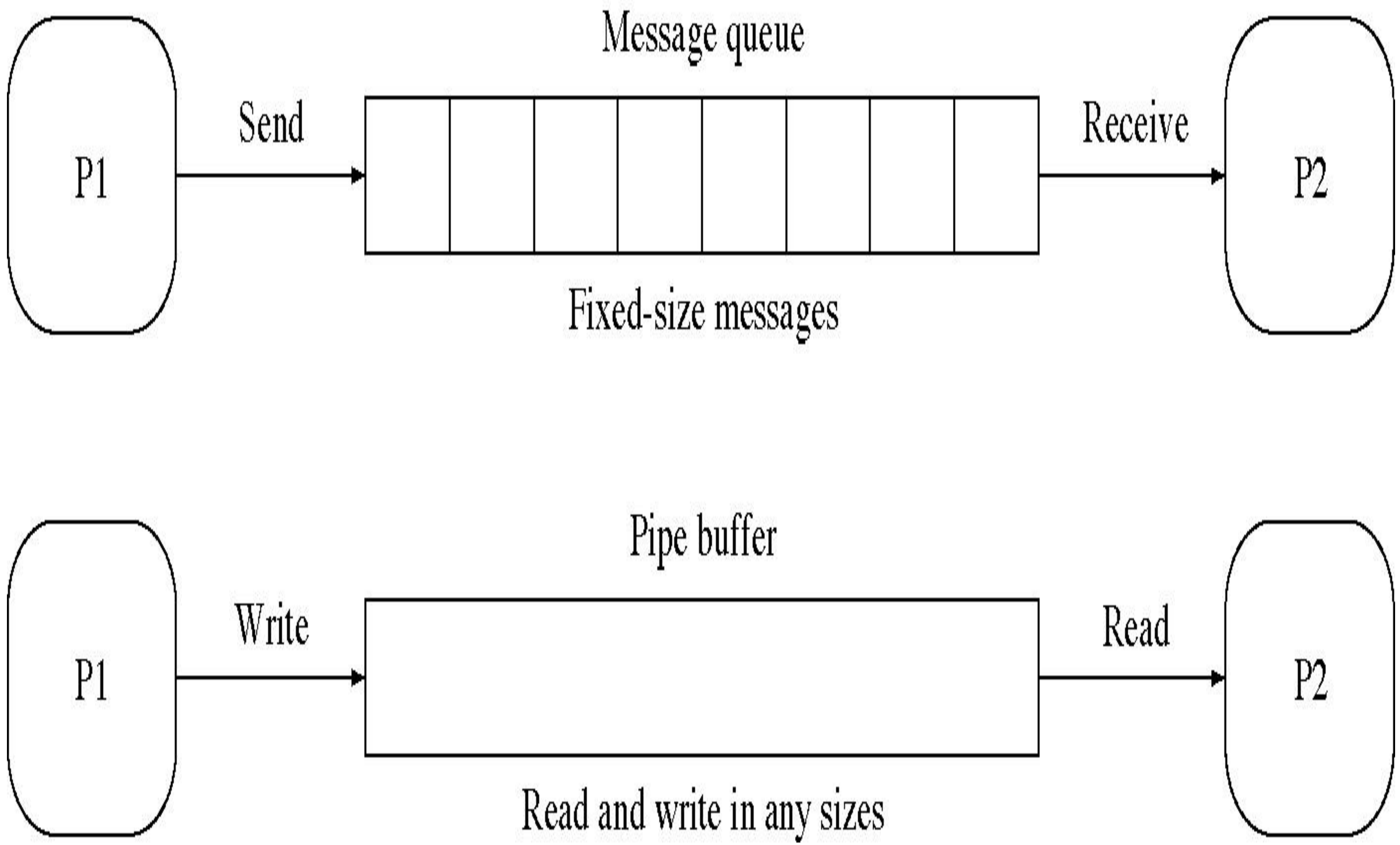


Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



Messages and Pipes Compared



Message Passing

- Message passing is a general method used for IPC:
  - for processes inside the same computer.
  - for processes in a networked/distributed system.
- In both cases, the process may or may not be blocked while sending a message or attempting to receive a message.

Synchronization in message passing (1)

- Message passing may be blocking or non-blocking.
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization in message passing (2)

- For the sender: it is more natural not to be blocked after issuing send:
  - can send several messages to multiple destinations.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails).
- For the receiver: it is more natural to be blocked after issuing receive:
  - the receiver usually needs the information before proceeding.
  - but could be blocked indefinitely if sender process fails before send.

Synchronization in message passing (3)

- Hence other possibilities are sometimes offered.
- Example: blocking send, blocking receive:
  - both are blocked until the message is received.
  - occurs when the communication link is unbuffered (no message queue).
  - provides tight synchronization (*rendezvous*).

Synchronization in message passing (4)

- There are really 3 combinations here that make sense:

1. Blocking send, Blocking receive
2. Nonblocking send, Nonblocking receive
3. Nonblocking send, Blocking receive – most popular – example:
    - Server process that provides services/resources to other processes. It will need the expected information before proceeding.

IPC Requirements

- If *P* and *Q* wish to communicate, they need to:
  - establish communication link between them.
  - exchange messages via send/receive.
- Implementation of communication link:
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- 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?

Link Capacity ─ Buffering

- Queue of messages attached to the link; implemented in one of three ways:

1. Zero capacity – 0 messages
   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 waits.

Direct/Indirect Communication

- Direct communication:
  - when a specific process identifier is used for source/destination.
  - but it might be impossible to specify the source ahead of time (e.g., a print server).
- Indirect communication (more convenient):
  - messages are sent to a shared mailbox which consists of a queue of messages.
  - senders place messages in the mailbox, receivers pick them up.

Direct Communication

- Processes must name each other explicitly:
  - **send**(*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from 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 (1)

- 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.

Indirect Communication (2)

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

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

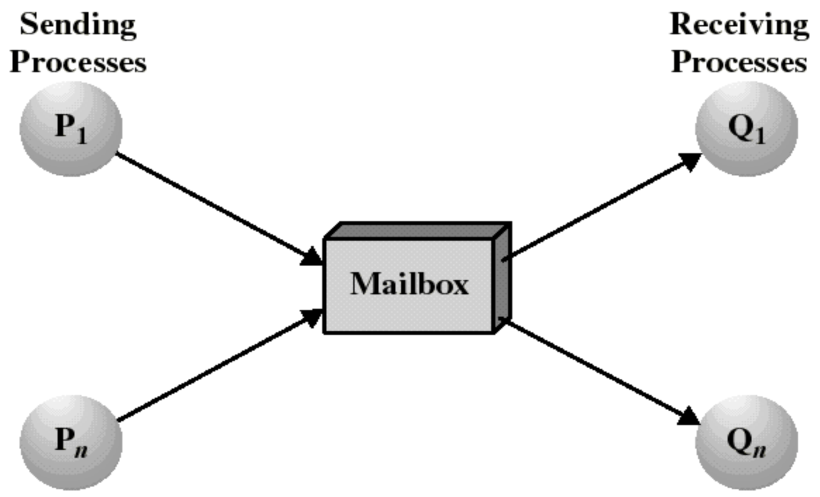
receive(*A, message*) – receive a message from mailbox A.

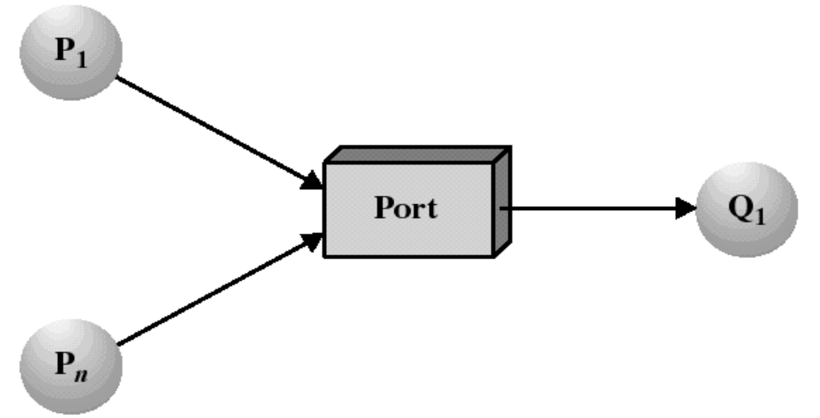
Indirect Communication (3)

- 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:
  - 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.

Mailboxes and Ports

- A mailbox can be private to one sender/receiver pair.
- The same mailbox can be shared among several senders and receivers:
  - the OS may then allow the use of message types (for selection).
- Port: is a mailbox associated with one receiver and multiple senders
  - used for client/server applications: the receiver is the server.





Ownership of ports and mailboxes

- A port is usually own and created by the receiving process.
- The port is destroyed when the receiver terminates.
- The OS creates a mailbox on behalf of a process (which becomes the owner).
- The mailbox is destroyed at the owner's request or when the owner terminates.

Mutual Exclusion – Message Passing

- create a mailbox ***mutex*** shared by n processes.
- send() is non-blocking.
- receive() blocks when ***mutex*** is empty.
- Initialization: send(***mutex***, “go”);
- The first Pi who executes receive() will enter CS. Others will be blocked until Pi resends msg.

```
Process Pi:
var msg: message;
repeat
   receive(mutex,msg);
   CS
   send(mutex,msg);
   RS
forever
```

Bounded-Buffer – Message Passing

- The producer place items (inside messages) in the mailbox ***mayconsume***.
- ***mayconsume*** acts as our buffer: consumer can consume item when at least one message present.
- Mailbox ***mayproduce*** is filled initially with k null messages (k= buffer size).
- The size of ***mayproduce*** shrinks with each production and grows with each consumption.
- Solution can support multiple producers/consumers.

Bounded-Buffer – Message Passing

```
Producer:
var pmsg: message;
repeat
   receive(mayproduce, pmsg);
   pmsg := produce();
   send(mayconsume, pmsg);
forever

Consumer:
var cmsg: message;
repeat
   receive(mayconsume, cmsg);
   consume(cmsg);
   send(mayproduce, null);
forever
```

P/C Problem with Message Passing (1)

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                         /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}
```

P/C Problem with Message Passing (2)

```
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);                   /* get message containing item */
        item = extract_item(&m);                 /* extract item from message */
        send(producer, &m);                      /* send back empty reply */
        consume_item(item);                      /* do something with the item */
    }
}
```

Examples of IPC Systems – POSIX

- POSIX Shared Memory example
- Process first creates shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(segment_id, NULL, 0);
```

- Now the process could write to the shared memory

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

- When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

- Now process can remove the shared memory segment

```
shmdt(shared_id, IPC_RMID, NULL);
```

Examples of IPC Systems – Mach

- Mach communication is message based:
  - Even system calls are messages.
  - Each task gets two mailboxes at creation:
    Kernel and Notify.
  - Only three system calls needed for message transfer:

`msg_send(), msg_receive(), msg_rpc()`

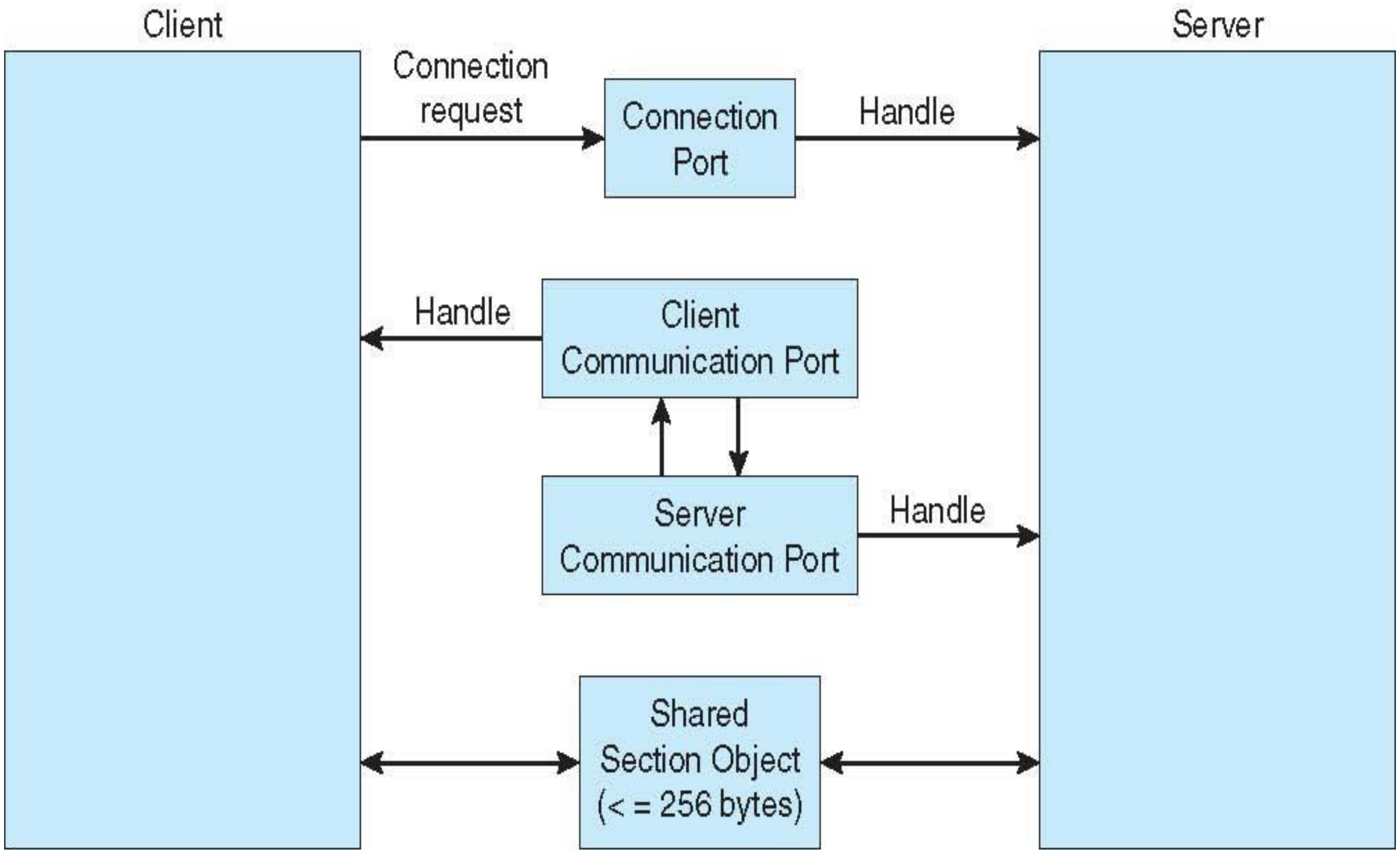
  - Mailboxes needed for communication, created via

`port_allocate()`

Examples of IPC Systems – Windows XP

- Message-passing centric via LPC facility:
  - Only works between processes on the same system.
  - Uses ports (like mailboxes) to establish and maintain communication channels.
  - Communication works as follows:
    - The client opens a handle to the subsystem’s connection port object.
    - The client sends a connection request.
    - The server creates two private communication ports and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

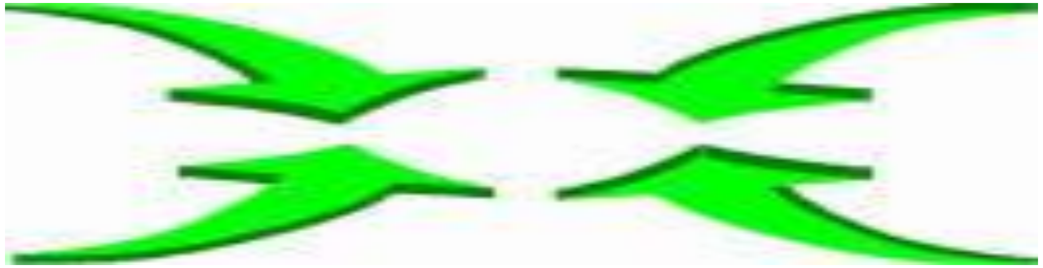
Local Procedure Calls in Windows XP



Communications in Client-Server Systems

- There are various mechanisms:

1. Pipes
2. Sockets (Internet)
3. Remote Procedure Calls (RPCs)
4. Remote Method Invocation (RMI, Java)



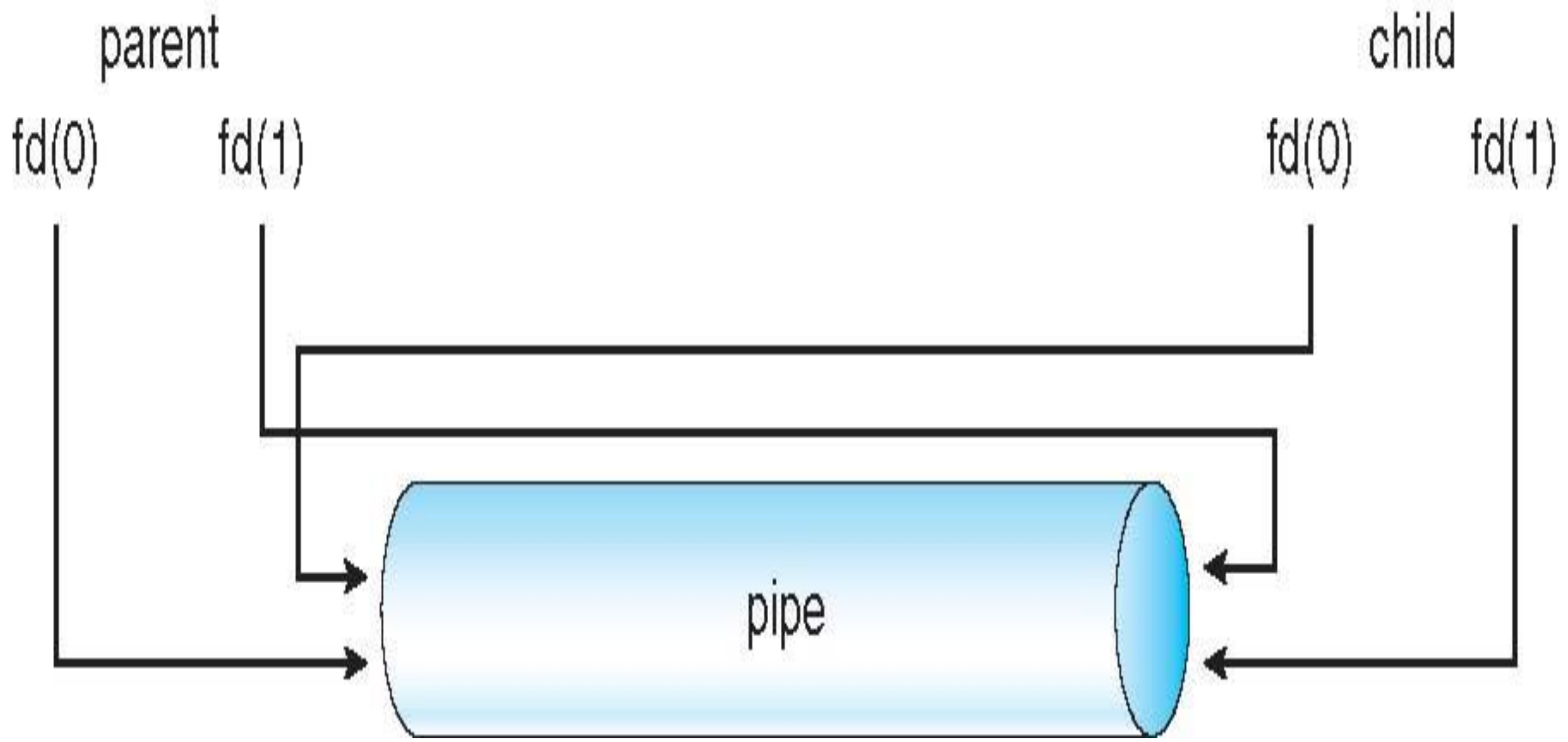
Pipes

- Acts as a conduit allowing two processes to communicate.
- Some 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

- **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.

Ordinary Pipes



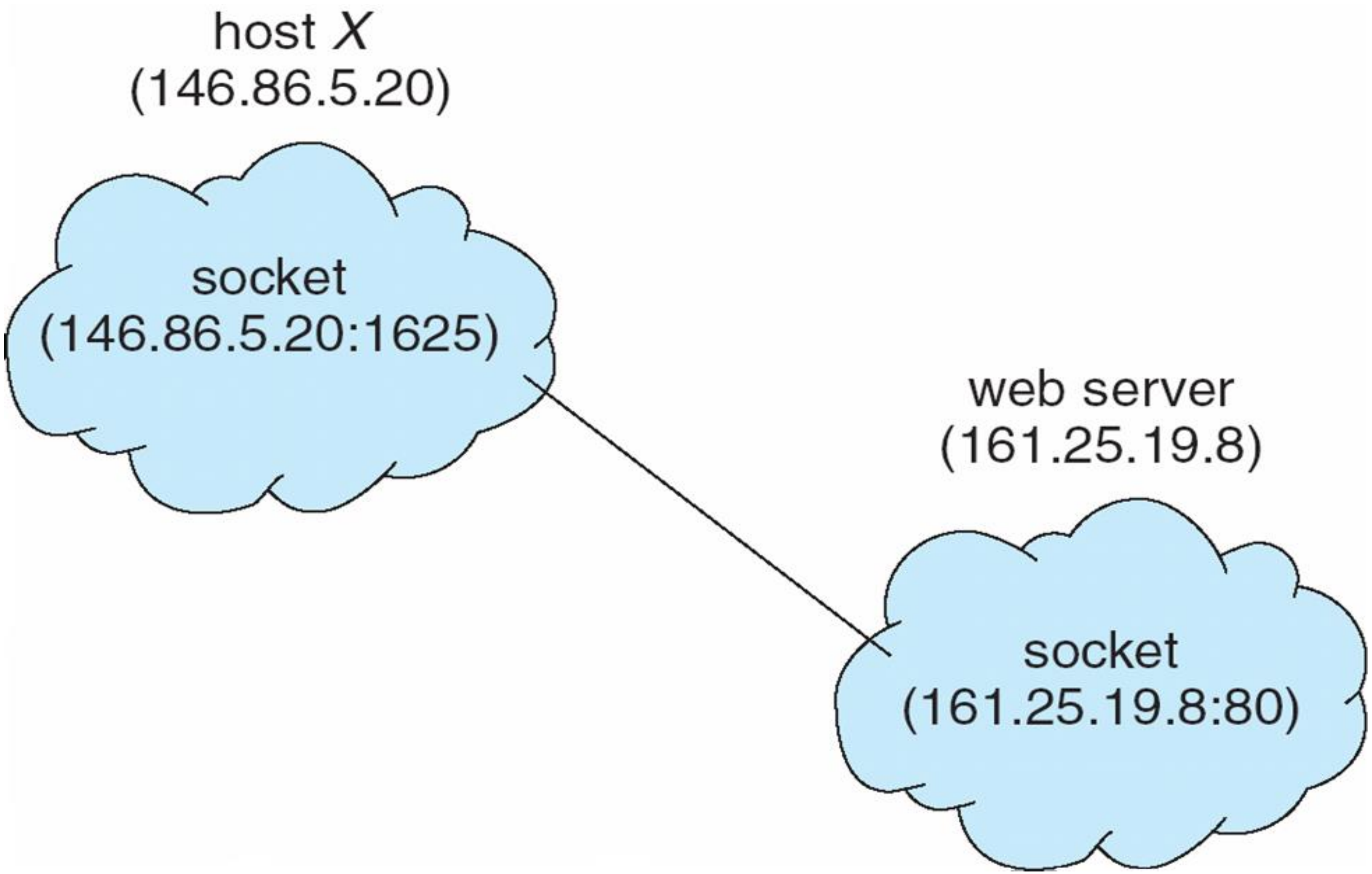
Named Pipes

- 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.

Sockets

- A socket is defined as an *endpoint for communication.*
- Concatenation of IP address and port.
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8.**
- Communication consists between a pair of sockets.

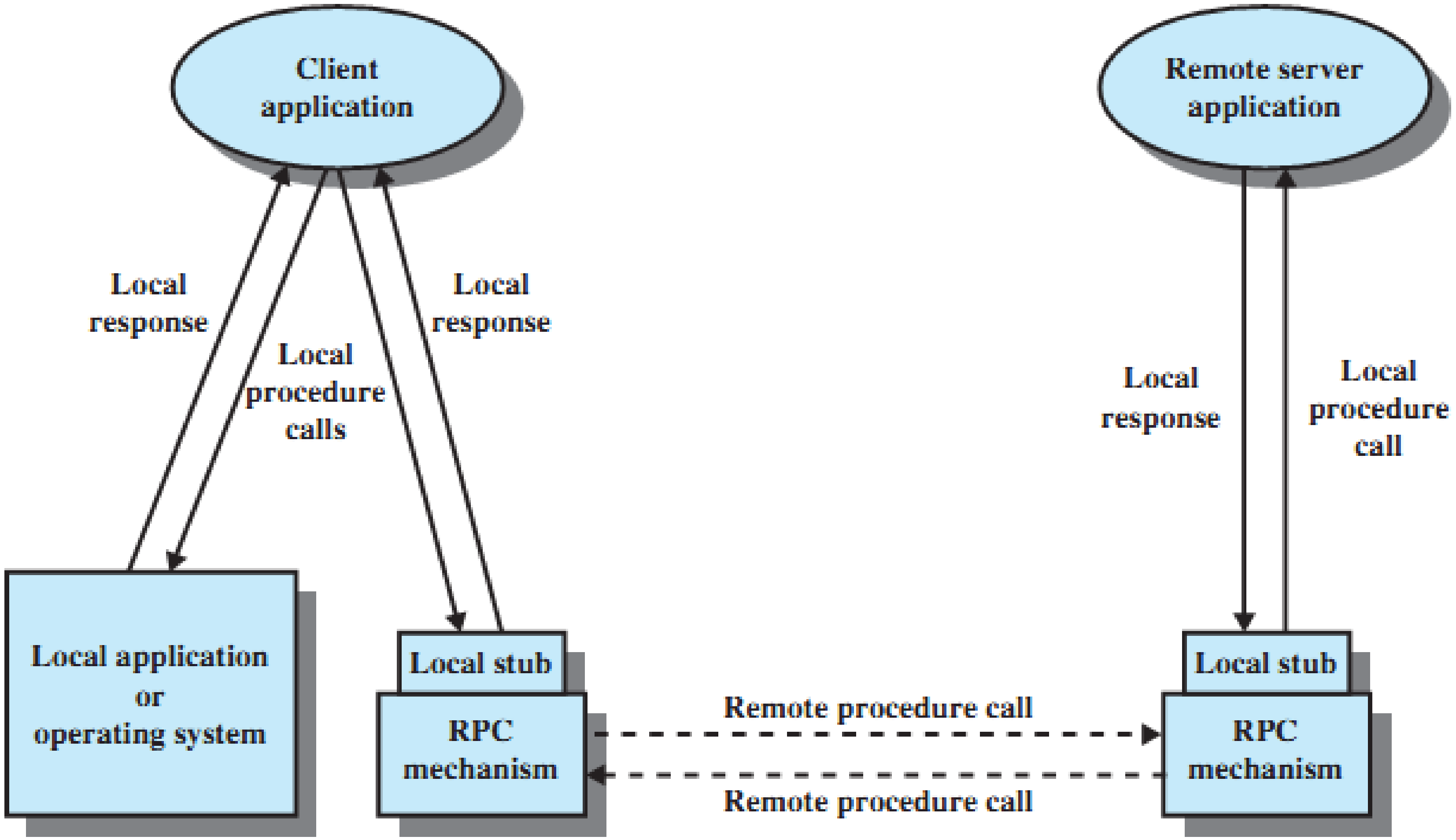
Socket Communication



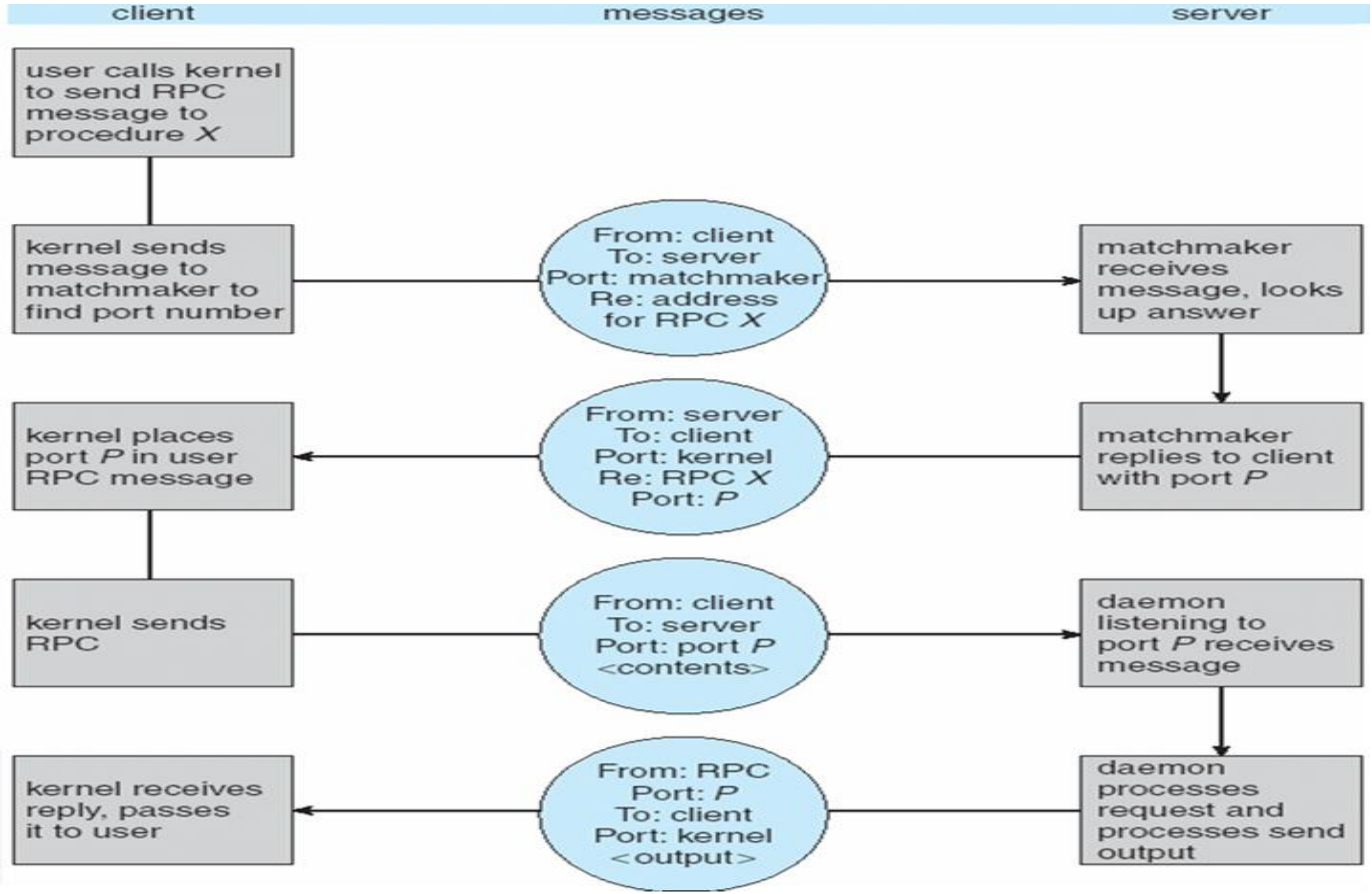
Remote Procedure Calls (RPCs)

- RPC abstracts a Local Procedure Call (LPC) between processes on a networked system.
- **Stubs** – client-side proxy for the actual procedure existing on the server.
- The client-side stub locates the server and *marshals* the parameters.
- The server-side stub/skeleton receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
- Vice versa happens on the opposite direction.

Remote Procedure Call Mechanism

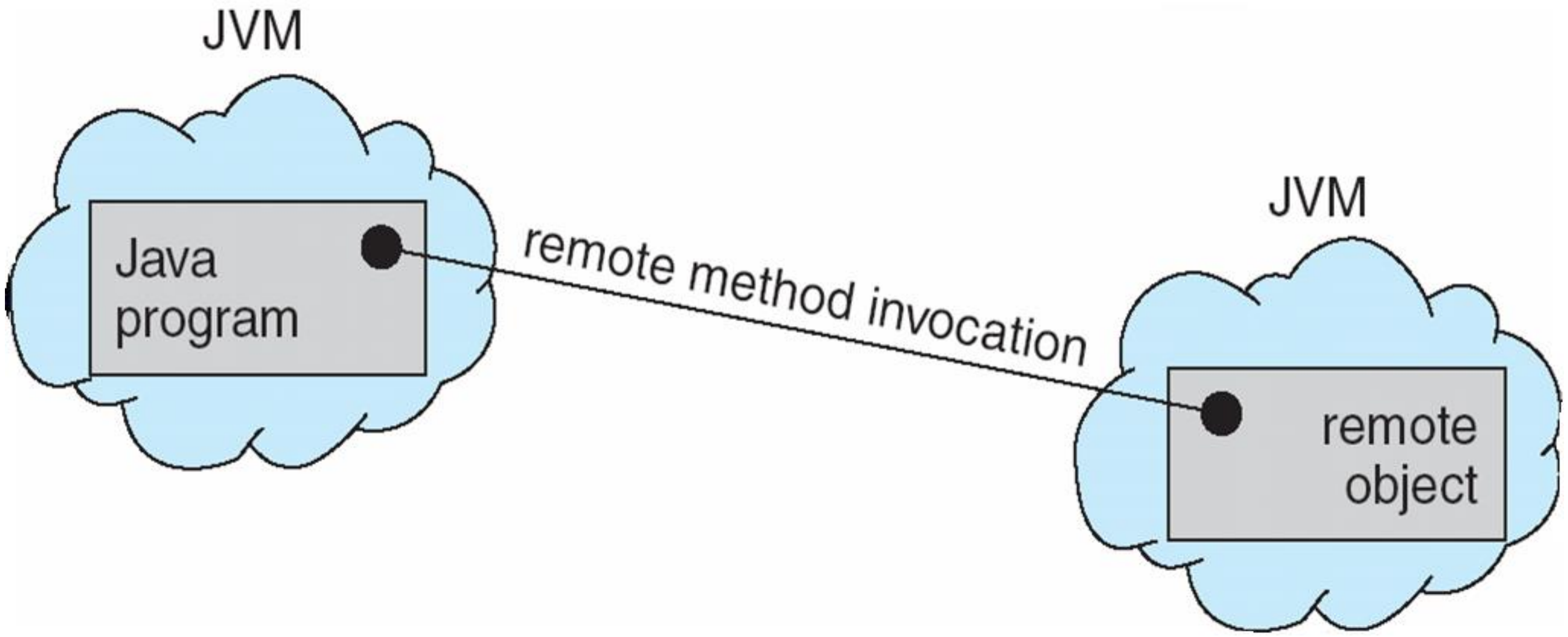


Execution of RPC

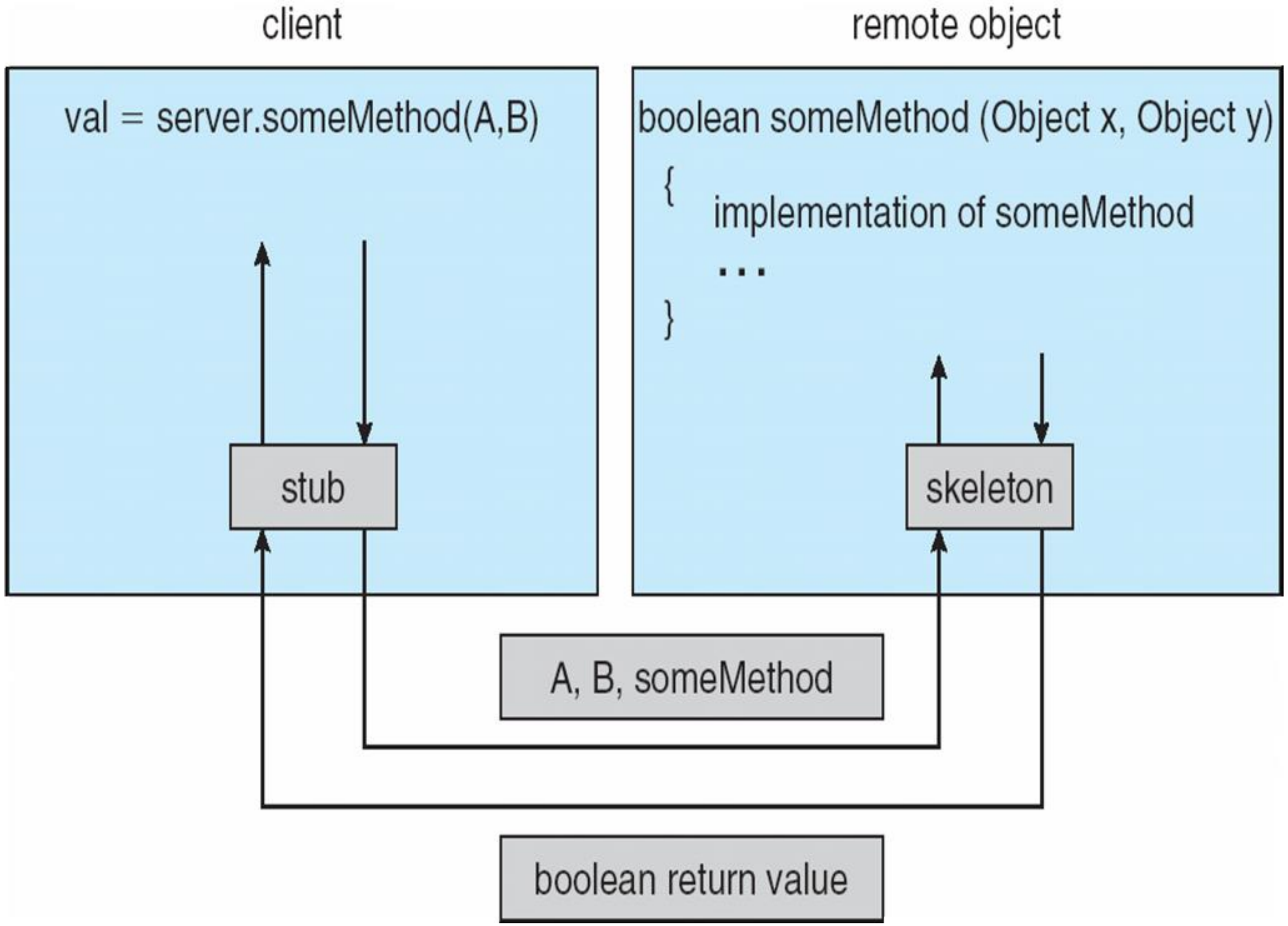


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



(Un)Marshalling Parameters



Python Implementation

- https://docs.python.org/3.4/library/ipc.html
- https://docs.python.org/3/library/ipc.html

Passing Messages to Processes (Queue)

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print 'Doing something fancy in %s for %s!' % (proc_name, self.name)


def worker(q):
    obj = q.get()
    obj.do_something()


if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

```
$ python multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

Signaling between Processes

```python
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print 'wait_for_event: starting'
    e.wait()
    print 'wait_for_event: e.is_set()->', e.is_set()

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print 'wait_for_event_timeout: starting'
    e.wait(t)
    print 'wait_for_event_timeout: e.is_set()->', e.is_set()


if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name='block',
                                 target=wait_for_event,
                                 args=(e,))
    w1.start()

    w2 = multiprocessing.Process(name='non-block',
                                 target=wait_for_event_timeout,
                                 args=(e, 2))
    w2.start()

    print 'main: waiting before calling Event.set()'
    time.sleep(3)
    e.set()
    print 'main: event is set'
```

```
$ python -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```

Synchronizing Processes

```python
import multiprocessing
import time

def stage_1(cond):
    """perform first stage of work, then notify stage_2 to continue"""
    name = multiprocessing.current_process().name
    print 'Starting', name
    with cond:
        print '%s done and ready for stage 2' % name
        cond.notify_all()

def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print 'Starting', name
    with cond:
        cond.wait()
        print '%s running' % name

if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1', target=stage_1, args=(condition,))
    s2_clients = [
        multiprocessing.Process(name='stage_2[%d]' % i, target=stage_2, args=(condition,))
        for i in range(1, 3)
        ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

```
$ python multiprocessing_condition.py

Starting s1
s1 done and ready for stage 2
Starting stage_2[1]
stage_2[1] running
Starting stage_2[2]
stage_2[2] running
```

Processes Pool

```python
import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print 'Starting', multiprocessing.current_process().name

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input   :', inputs

    builtin_outputs = map(do_calculation, inputs)
    print 'Built-in:', builtin_outputs

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process,
                                )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join()  # wrap up current tasks

    print 'Pool    :', pool_outputs
```

```
$ python multiprocessing_pool.py

Input   : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting PoolWorker-11
Starting PoolWorker-12
Starting PoolWorker-13
Starting PoolWorker-14
Starting PoolWorker-15
Starting PoolWorker-16
Starting PoolWorker-1
Starting PoolWorker-2
Starting PoolWorker-3
Starting PoolWorker-4
Starting PoolWorker-5
Starting PoolWorker-8
Starting PoolWorker-9
Starting PoolWorker-6
Starting PoolWorker-10
Starting PoolWorker-7
Pool    : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Client/Server with Sockets

```
from multiprocessing.connection import Client

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')
conn.send('close')
# can also send arbitrary objects:
# conn.send(['a', 2.5, None, int, sum])
conn.close()
```

```
from multiprocessing.connection import Listener

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'
listener = Listener(address, authkey='secret password')
conn = listener.accept()
print 'connection accepted from', listener.last_accepted
while True:
    msg = conn.recv()
    # do something with msg
    if msg == 'close':
        conn.close()
        break
listener.close()
```

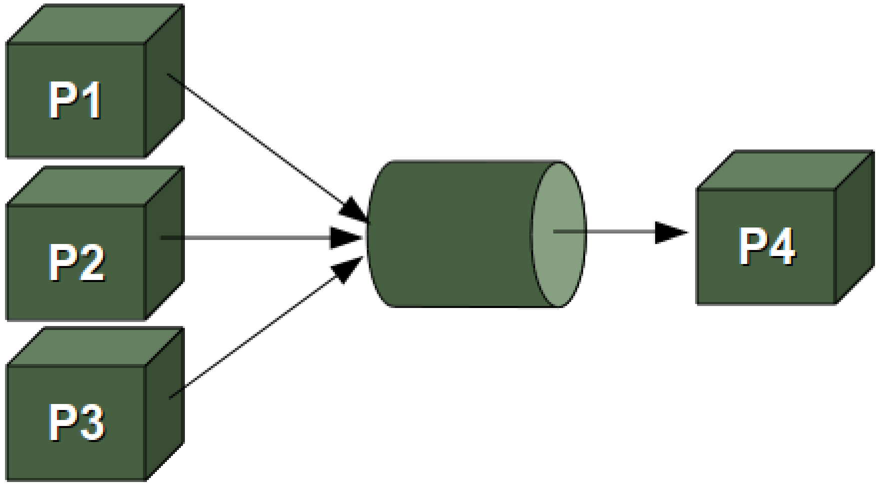
Named Pipe, FIFOs

```
import os, time, sys
pipe_name = 'pipe_test'

def child( ):
    pipeout = os.open(pipe_name, os.O_WRONLY)
    counter = 0
    while True:
        time.sleep(1)
        os.write(pipeout, 'Number %03d\n' % counter)
        counter = (counter+1) % 5

def parent( ):
    pipein = open(pipe_name, 'r')
    while True:
        line = pipein.readline()[:-1]
        print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time( ))

if not os.path.exists(pipe_name):
    os.mkfifo(pipe_name)
pid = os.fork()
if pid != 0:
    parent()
else:
    child()
```



SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY