Inter-Process Communications (IPCs)

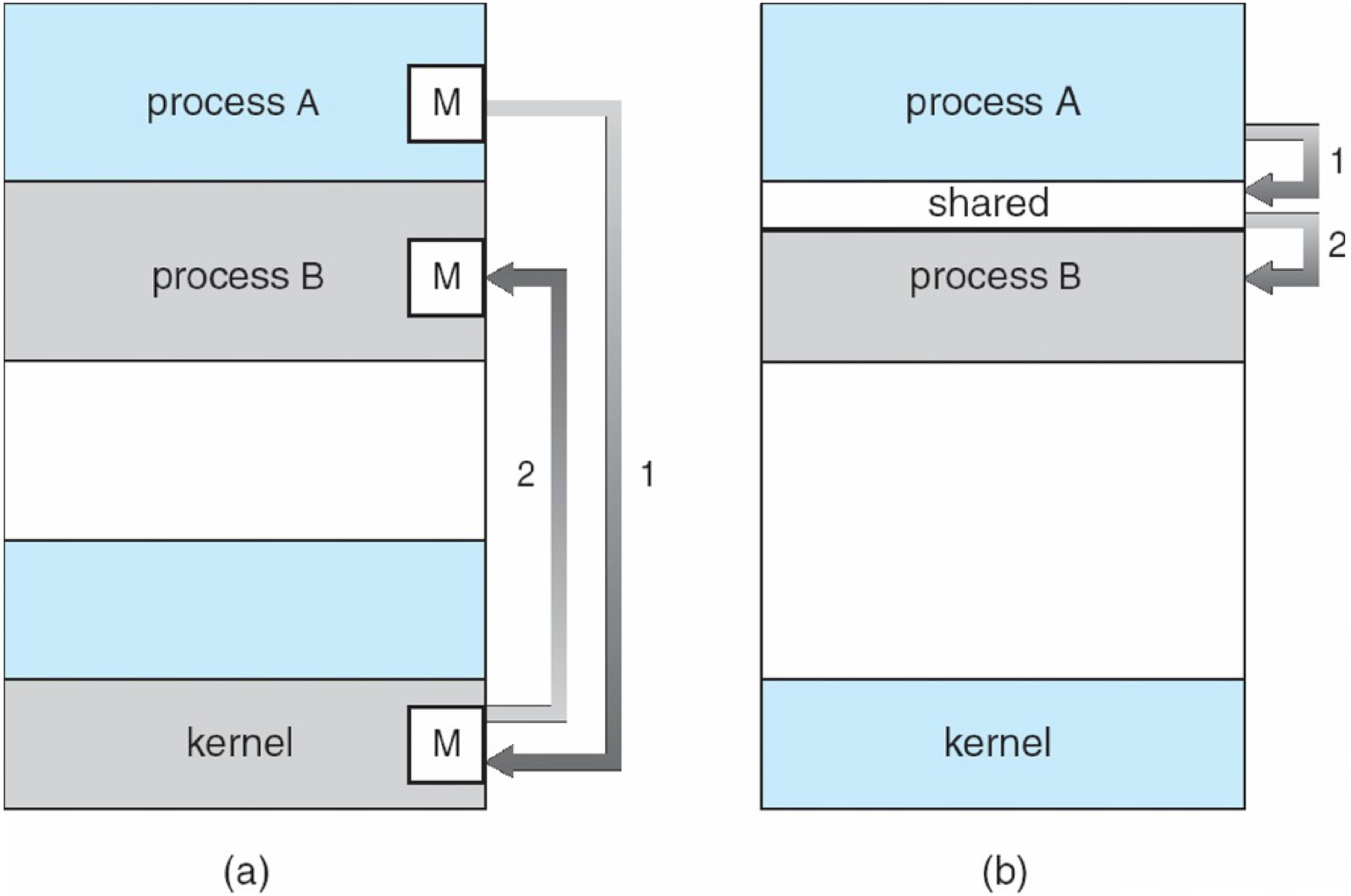
ICS332

Operating Systems

Communicating Processes

- Processes within a host may be **independent** or **cooperating**
- Reasons for cooperating processes:
  - Information sharing
    - e.g., Coordinated access to a shared file
  - Computation speedup
    - e.g., Each process uses a different core (more likely done w/ threads)
  - Modularity
    - e.g., Systems designed as sets of processes are modular because one process can be easily replaced by another
  - Convenience
    - Some tasks are expressed naturally as sets of processes
- The means of communication for cooperating processes is called **Interprocess Communication** (**IPC**)
- Two broad models of IPC
  - Shared memory
  - Message passing

Communication Models



message passing

shared memory

Communication Models

- Mainstream OSes (Lin, Win, Mac) implement both models
- Message-passing
  - useful for exchanging small amounts of data
  - simple to implement in the OS
  - sometimes cumbersome for the user as code is sprinkled with send/recv operations
  - high-overhead: one syscall per communication operation
- Shared memory
  - low-overhead: a few syscalls initially, and then none
  - more convenient for the user since we’re used to simply reading/writing from/to RAM
  - more difficult to implement in the OS

Shared Memory

- Processes need to establish a shared memory region
  - One process creates a shared memory segment
  - Processes can then “attach” it to their address spaces
    - Note that this is really contrary to the memory protection idea central to multi-programming!
- Processes communicate by reading/writing to the shared memory region
  - They are responsible for not stepping on each other’s toes
  - The OS is not involved at all
- The textbook has a producer/consumer example, which you must read (Section 3.4.1)
  - It’s in C, but very Java-like
  - Processes read/write data in a shared buffer
  - We’ll talk about producer/consumer again

Example: POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment

```
id = shmget(IPC_PRIVATE, size, IPC_R | IPC_W);
```

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

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

  - Now the process can write to the shared memory

```
sprintf(shared_memory, “hello”);
```

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

```
shmdt(shared_memory);
```

  - Complete removal of the shared memory segment is done with

```
shmctl(id, IPC_RMID, NULL);
```

- See posix_shm_example.c

Example: POSIX Shared Memory

- Question: How do processes find out the ID of the shared memory segment?
- In posix_shm_example.c, the id is created before the fork() so that both parent and child know it
  - How convenient!
- There is no general solution
  - The id could be passed as a command-line argument
  - The id could be stored in a file
  - Better: one could use message-passing to communicate the id!
- On a system that supports POSIX, you can find out the status of IPCs with the ‘ipcs -a’ command
  - run it as root to be able to see everything
  - you’ll see two other forms of ipcs: Message Queues, and Semaphores

It all seems cumbersome

- The code for using shm ipcs is pretty cumbersome
  - The way to find out the id of the memory segment is clunky, at least
- This is perhaps not surprising given that we’re breaking one of the fundamental abstractions provided by the OS: memory isolation
  - We’ll see how memory isolation is implemented and how it can be broken for sharing memory between processes in the second part of the semester
- Nowadays shm-type code is not very common, which is probably a good thing
  - But processes still share memory under the cover (e.g., code segments for standard library functions)
- Sharing memory among multiple running context is done using threads, as we’ll see in the next lecture

All of the power of shm stuff, none of the inconvenience

Message Passing

- With message passing, processes do not share any address space for communicating
  - So the memory isolation abstraction is maintained
- Two fundamental operations:
  - send: to send a message (i.e., some bytes)
  - recv: to receive a message (i.e., some bytes)
- If processes P and Q wish to communicate they
  - establish a communication “link” between them
    - This “link” is an abstraction that can be implemented in many ways
      - even with shared memory!!
  - place calls to send() and recv()
  - optionally shutdown the communication “link”
- Message passing is key for distributed computing
  - Processes on different hosts cannot share physical memory!
- But it is also very useful for processes within the same host

Implementing Message-Passing

- Let’s pretend we’re designing a kernel, and let’s pretend we have to design the message-passing system calls
- Let’s do this now to see how simple it can be
  - I am going to show really simple, unrealistic pseudo-code
- Let’s say we don’t want an explicit link establishing call to keep things simple
- We have to implement two calls
  - send(Q, message): send a message to process Q
  - recv(Q, message): recv a message from process Q

Implementing Message-Passing

- We’ll implement communication between processes as a set of Message objects, say, in a MessageQueue class
- We need to keep track of all MessageQueue objects so that when process P wants to talk to process Q, we can find their MessageQueue object
- Let’s keep track of MessageQueue objects in a MessageQueueManager singleton (indexed by the PID of P and Q)
- The MessageQueueManager, MessageQueue, and Message objects are stored in the memory of the kernel
  - Therefore, they can’t get too big, and a real implementation would have to return an “out of memory” error if we use too many bytes (e.g., many large messages sent but not received)

Implementing Message-Passing

```
class ProcessImplementingMessagePassing extends Process {
/* Send a message from this process (P) to process Q */
public void send(int pidProcessQ, Message message) {
    int pidProcessP = getMyPid();
    // Get the Queue associated to (pidProcessP, pidProcessQ)
    // (getQueue() creates the Queue if it doesn't exist
    MessageQueue q = MessageQueueManager.getQueue(pidProcessP,pidProcessQ);
    q.putMessage(message);
}
/* Receive a message sent from process Q (identified by pidProcessQ)
public Message recv(int pidProcessQ) {
    int pidProcessP = getMyPid();
    MessageQueue q = MessageQueueManager.getQueue(pidProcessP,pidProcessQ);
    return q.getMessage();
}

} // class ProcessImplementingMessagePassing
```

Implementing Message-Passing

```
public void send(int pidProcessQ, Message message) {
    int pidProcessP = getMyPid();
    // Get the Queue associated to (pidProcessP, pidProcessQ)
    // (getQueue() creates the Queue if it doesn't exist
    MessageQueue q = MessageQueueManager.getQueue(pidProcessP,pidProcessQ);
    q.putMessage(message); // Should this make a copy of the message?
}

public Message recv(int pidProcessQ) { // what if I want to receive from anybody?
    int pidProcessP = getMyPid();
    MessageQueue q = MessageQueueManager.getQueue(pidProcessP,pidProcessQ);
    return q.getMessage(); // should block if q is empty?
}
```

Message Passing Design Decisions

- There are many possible design decisions
  - Fixed- or variable-length messages
  - Can a link be associated to more than two processes?
    - Not in our pseudo-implementation
  - Can there be more than one link between two processes?
    - Not in our pseudo-implementation
  - Is a link uni- or bi-directional?
    - In our pseudo-implementation: unidirectional
  - etc.
- Let's look at 3 questions:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Direct Communication

- That’s what our pseudo-implementation did
- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q*) – 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
- Asymmetric communication “challenge”:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*&Who*) – receive a message from any process, whose identity is stored in variable Who when the call returns

Indirect Communication

- Messages transit through mailboxes (or “ports” or “doors”)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of the 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
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives:
  - **A = createMailbox()**
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A*) – receive a message from mailbox A

Indirect Communication

- The mailbox sharing issue:
  - 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 mailbox 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
    - Perhaps notify the sender of who the receiver was

Word of Wisdom

- Designing systems requires spending (a lot of) time discussing such issues

  Decision driven by constraints and requirements
- It turns out that the definition of abstractions (semantics and APIs) always has deep implications
  - Many of which are difficult to foresee
  - Many of which cause disasters
- Being good at designing good abstractions is a very valuable skill
  - Comes w/ experience and knowledge of existing systems

Synchronous/Asynchronous

- The terms blocking/non-blocking and synchronous/asynchronous are typically used interchangeably
  - In some contexts, subtle differences are made, but we can ignore them in this course
- Message passing may be either blocking or non-blocking
- **Blocking = synchronous (in OS context)**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
  - When both are blocking, the operation is called a rendez-vous communication style
- **Non-blocking = asynchronous (in OS context)**
  - **Non-blocking** send has the sender send the message and continue
    - With the option to check on status later (“was my message received?”)
  - **Non-blocking** receive has the receiver receive a valid message or null
    - With the option to block

Buffering

- While messages are in transit, they reside “in the link” (e.g., our MessageQueue object)
- There are three typical message queue implementations
  - Zero-capacity
    - There can be no waiting message
    - The sender is blocked
    - This enforces a “rendez-vous”
  - Bounded capacity
    - At most n messages can reside in the queue
      - Or n message bytes
    - If the queue is full, then the sender must block
  - Unbounded capacity
    - The sender never blocks
      - There should never be anything truly unbounded though

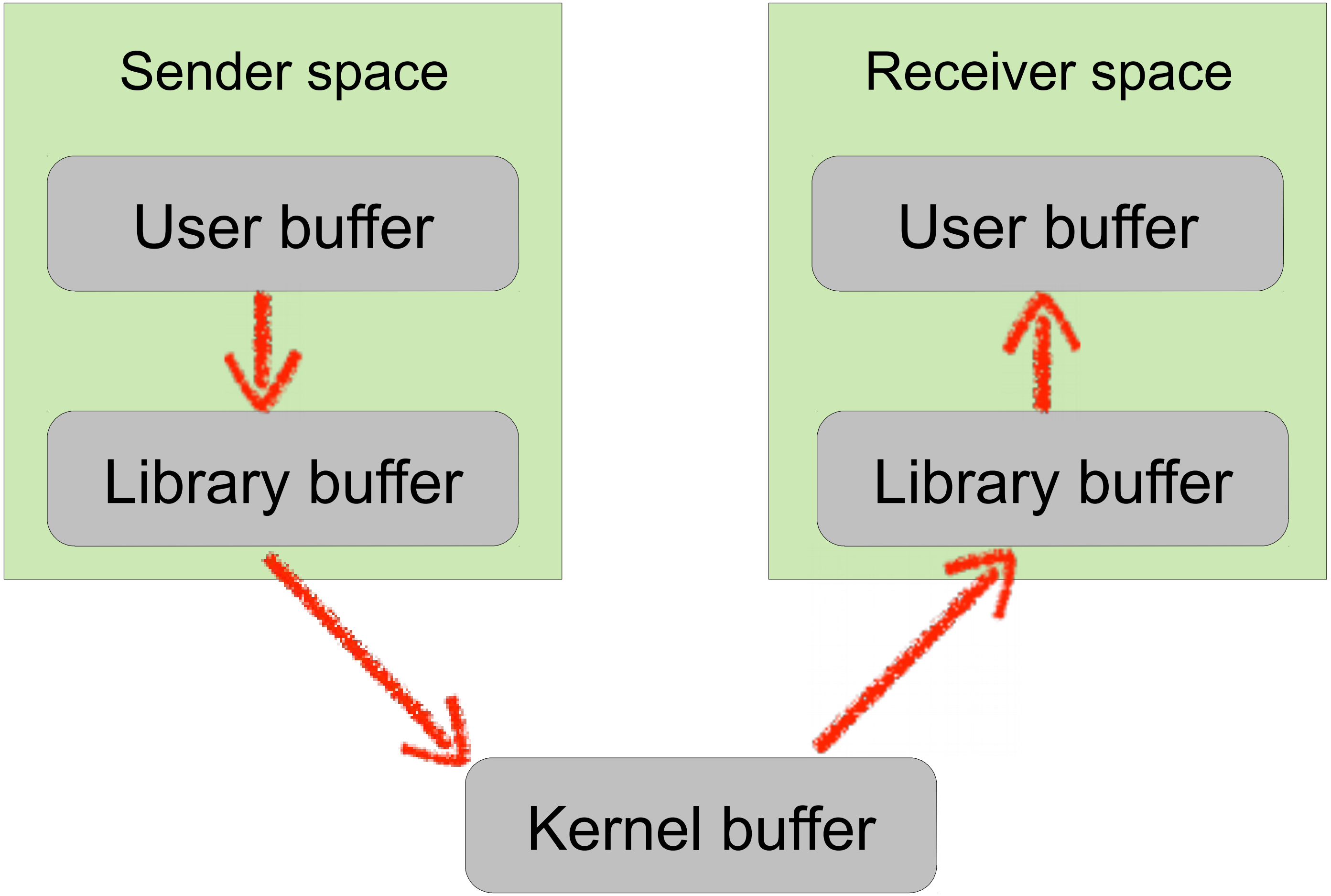
Example: Mach Message Passing

- Section 3.5.2 in the textbook goes through a description of mailbox-based message passing in the Mach kernel
  - It’s not difficult, but make sure you read it
- Essentially, it’s a message-passing system that makes particular choices regarding design decisions
- Consider the length/detail of a full description (already 2 pages what high-level overview in the book)
- Extra copies: big performance hit for message-passing
  - At a minimum: two copies
    - copy from user space to kernel space, and the reverse
  - Mach uses some sort of hidden shared memory implementation of message-passing to avoid the copies!
  - Looks a bit like the POSIX shm stuff
- In general, memory copies are performance killers

Why Memory Copies?

- Let's say you want to implement a message passing library that's convenient to use and that has the following semantics:
  - Once a send has been placed by a process, that process can safely overwrite the message that contains the data that was sent
    - No need for the user to keep wondering "has it been received yet and can I reuse/overwrite that memory?"
  - The send() function returns as soon as possible given the above semantic
    - The sender should do quick sends, and then move on to other work
- To do this, many memory copies may happen

Memory Copies Galore



Sender space

User buffer

Library buffer

Receiver space

User buffer

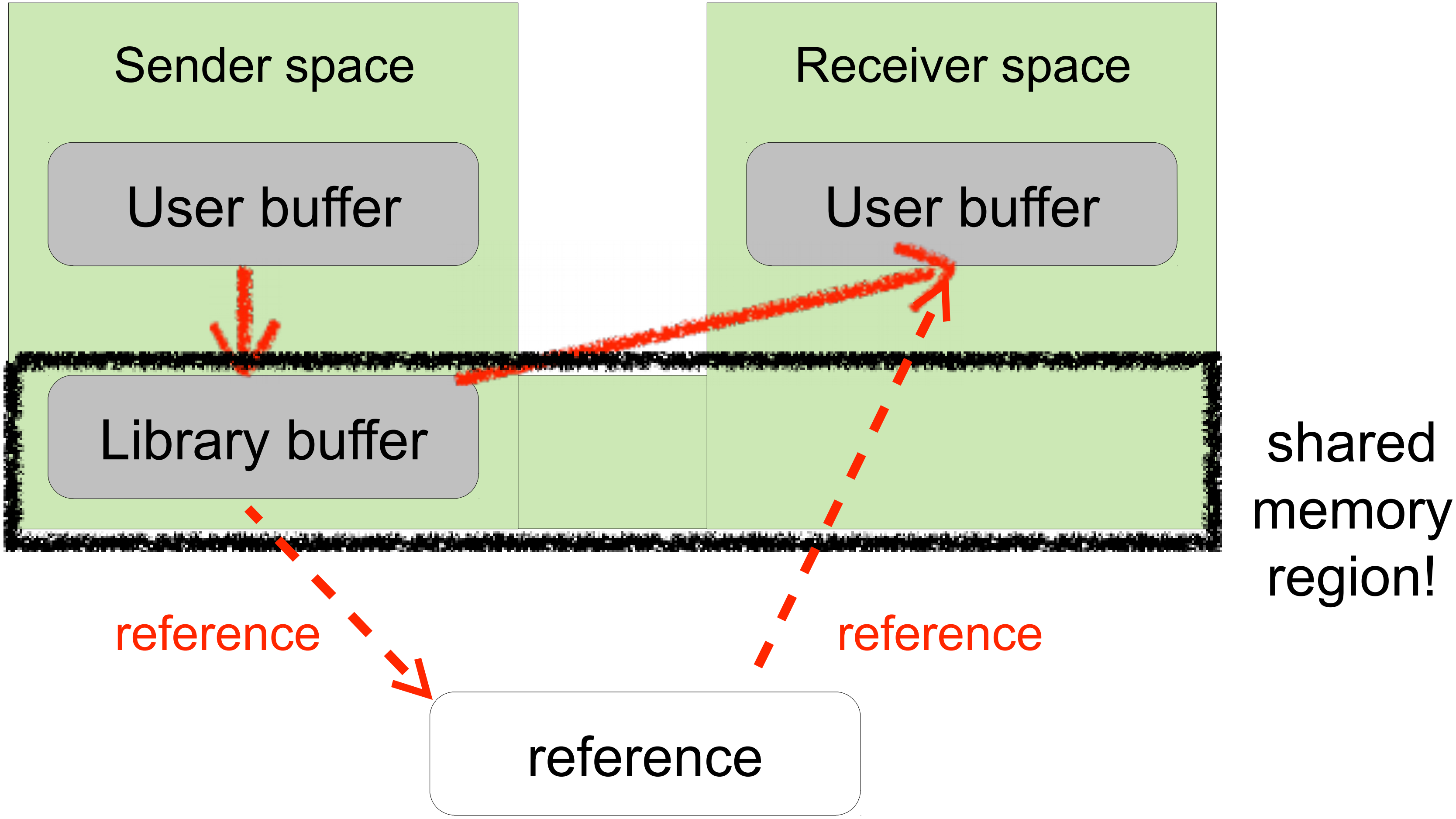
Library buffer

Kernel buffer

Reducing Memory Copies

- Reducing the number of memory copies is a well-known goal in system code
  - So-called “zero-copy” implementations
- In our example there are 4 memory copies
- The copies from user space to kernel space could be avoided
  - If the kernel provides a send/recv abstraction that does take only pointers, does not do any copy, and is simply told “here is a pointer to a message but I guarantee you that it won’t be overwritten/erased”, then we can have a different picture, assuming that a shared-memory region is available

Memory Copies Galore



Client-Server Communication

- Applications are often structured as sets of communication processes
  - Common across machines (Web browser and Web server)
  - But useful within a machine as well
- Let’s look at
  - Sockets
  - RPCs (Remote Procedure Calls)
  - LPCs (Local PC) in WinNT (renamed ALPC (Advanced LPC) from WinVista)
  - Java RMI
  - Pipes (not in book)
- Tons of other ones (named pipes, shared message queues, CORBA, Google Web Toolkit, Apache Thrift, ...)
  - The history of IPCs is huge and the number of IPC implementations/abstractions is staggering

Example: Sockets

- A socket is a data communication endpoint so that two processes (running on the same host for “Unix or IPC” sockets / fyi: on different hosts for “network” sockets) can communicate.
  - Socket = ip address + port number
- Sockets are typically used to communicate between two different hosts, but also work within a host
  - Most network communication in user programs is written on top of the socket abstraction
    - e.g., you’d find sockets in the code of a Web browser
- Section 3.6.1 describes Sockets
  - Something you’ll see in a networking course

Remote Procedure Calls

- So far, we've seen unstructured message passing
  - A message is just a sequence of bytes
  - It's the application's responsibility to interpret the meaning of those bytes
- RPC provides a procedure invocation abstraction across hosts
  - A "client" invokes a procedure on a "server", just as it invokes a local procedure
- The magic is done by a client stub, which is code that:
  - marshals arguments
    - Structured to unstructured, under the cover
  - sends them over to a server
  - wait for the answer
  - unmarshals the returned values
    - Unstructured to structured, under the cover
- A variety of implementations exists
- Section 3.6.2 in the textbook covers RPC

RPC Semantics

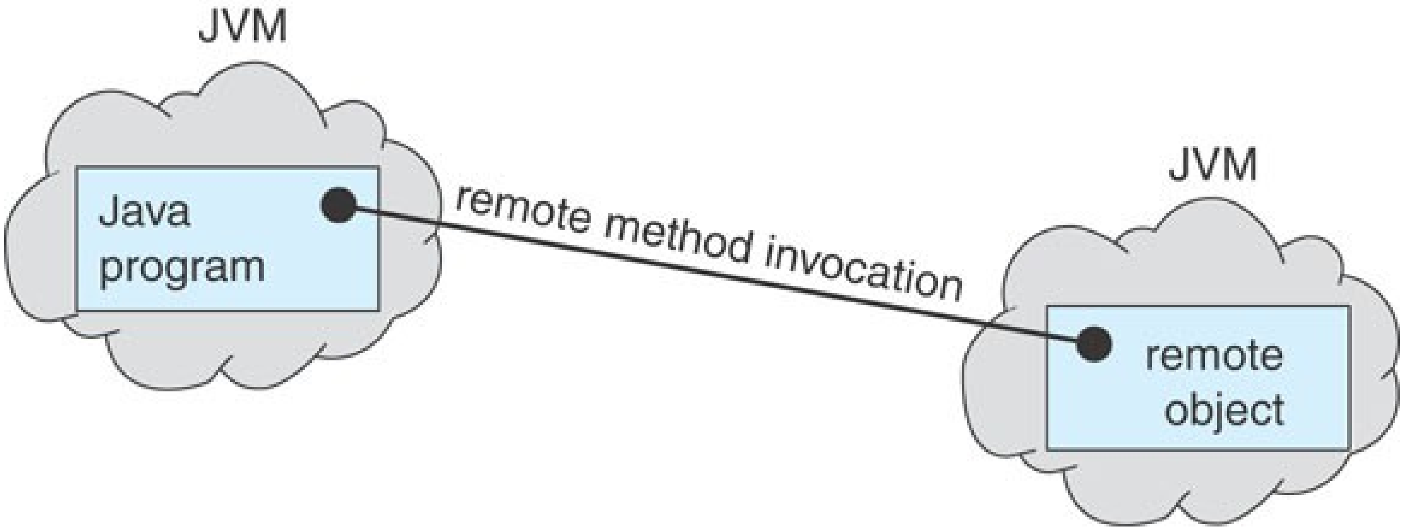
- One interesting issue: what happens if the RPC fails
  - standard procedure calls almost never fails
- Danger:
  - The RPC was partially executed
  - The RPC was executed multiple times due to retries that shouldn't have been attempted
- Weak (easy to implement) semantic: at most once
  - Server maintains a time-stamp of incoming messages
  - If a repeated message shows up, ignore it
  - The client can be overzealous with retries
  - But the server may never perform the work
- Strong (harder to implement) semantic: exactly once
  - The server must send an ack to the client saying "I've done it"
  - The client periodically retries until the ack is received

Local Procedure Calls in Win

- Windows XP uses an LPC mechanism for structured message passing between processes on the same host
  - Essentially like RPC, but just happens to be local, and therefore doesn’t go out to the network
  - Described in Section 3.5.2 / Undocumented by MS
- LPCs are not visible to the application program, but are hidden inside the code of the Win32 library
  - It’s something that system developers use, and that Win32 users use without knowing they do
- Like in Mach, a shared-memory trick is used to improve performance for large messages and avoid memory copies
  - The caller can request a shared memory region, in which messages will be stored/retrieved and not copied back and forth from user space to kernel space
    - This is obviously not possible with RPCs

Java RMI

- RMI is essentially “RPC in Java” in an object-oriented way
- A process in a JVM can invoke a method of an object that lives in another JVM



Java RMI

- The great thing about RMI is that method arguments are marshalled/unmarshalled for you by the JVM
- Objects are serialized and deserialized
  - via the java.io.*Serializable* interface
- RMI sends copies of local objects and references to remote objects
- See the books (and countless Java RMI tutorials) for how to do this
  - This will come in handy if you write distributed Java systems
- RMI hides most of the gory details of IPCs
  - More convenient, but not more “power” (i.e., you can do with Sockets everything you can do with RPC)

UNIX Pipes

- Pipes are one of the most ancient, yet simple and useful, IPC mechanisms provided by UNIX
  - They’ve also been available in MS-DOS from the beginning
- In UNIX, a pipe is mono-directional

  (Two named pipes (mkfifo) can be used for bidirectional communication)
- One talks of the write-end and the read-end of a pipe
- The “pipe” command-line feature, |, corresponds to a pipe
- The command “ls | grep foo” creates two processes that communicate via a pipe
  - The ls process writes on the write-end
  - The grep process reads on the read-end
- An arbitrary number of pipes can be created:
  - ls -R / | grep foo | grep -v bar | wc -l
- The book has C examples of how to use pipes (Section 3.6.3)

Java: Communication with an External OS Process

- Spawning external processes using the ProcessBuilder class
  - Has a constructor that takes a command and a list of arguments, just as if you were to run the command in a Shell's command line
  - Creates a Process object, that can be communicated via standard streams, which are used for IPC
- Let's look at ProcessBuilderExample.java
  - And find out more on your own through the JDK documentation

Java: Synchronous and Asynchronous I/O

- I/O implemented in java.io is synchronous
  - read(), readLine() wait until data is available for reading
  - At this point, I'll assume we're all familiar with java.io
- Synchronous I/O is simple to implement but
  - Difficult to avoid a process just "hanging": should I attempt to call readLine() knowing that I may get stuck in it for hours?
  - Difficult to get data from multiple streams concurrently: should I attempt to get data from stream A and get stuck there for 10 minutes when 1 second from now there could be data available from stream B?
- Asynchronous I/O is implemented in java.nio
  - Designed to provide lower-level access to I/O operations
  - Channel + Buffer replaces Stream
  - Selector for managing multiple Channels
  - This is what you should use for high-performance I/O

Signals

- Signals are a UNIX form of IPC: used to notify a process that some even has occurred
  - They are some type of high-level software interrupts
  - Windows emulates them with APCs (Asynchronous Procedure Calls)
- Example: on a Linux box, when you hit ^C, a SIGINT signal is sent to a process (e.g., the process that's currently running in your Shell)
- They can be used for IPCs and process synchronization, but better methods are typically preferred (especially with threads)
  - Signals and threads are a bit difficult to manage together
- Once delivered to a process, a signal must be handled
  - Default handler (e.g., ^C is handled by terminating)
  - The user can specify that a signal should be ignored or can provide a user-specified handler (not allowed for all signals)

Conclusion

- Communicating processes are the bases for many programs/services
- OSes provide two main ways for processes to communicate
  - shared memory
  - message-passing
- Each way comes with many variants and in many flavors
  - Sockets, RPCs, Pipes, LPCs, RMI, signals