Inter-Process Communications(IPCs)

Operating Systems

Wenbo Shen

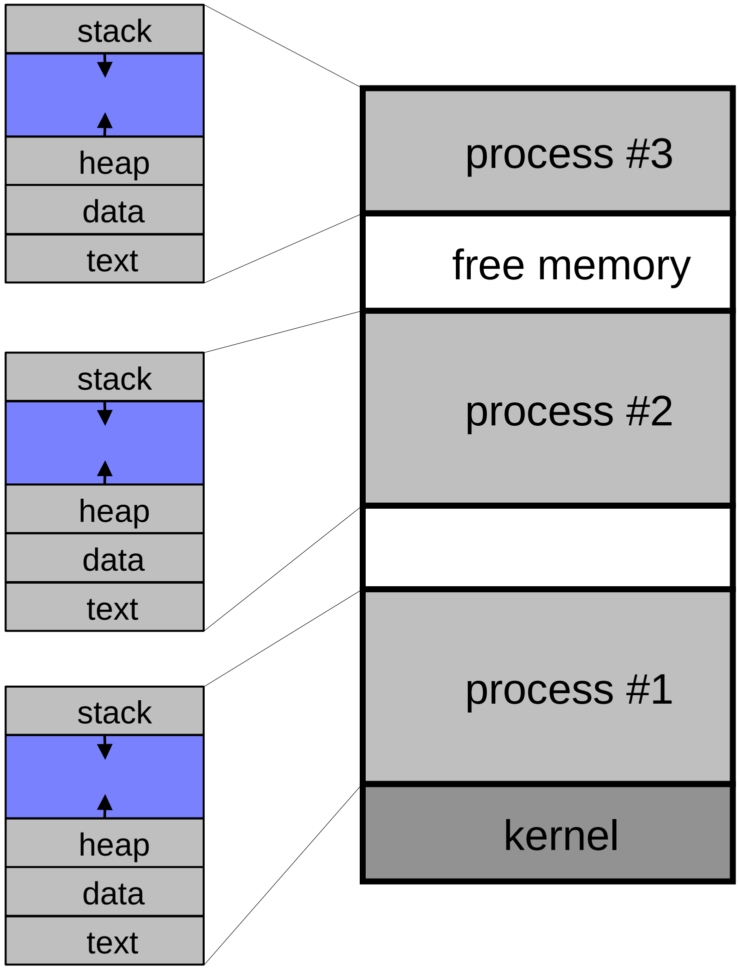
Some slides are from OSC and Henri Casanova@Hawaii

Inter-process Communication (IPC)

- 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., Multi-processing on the same task
  - Modularity
  - Convenience
- The means of communication for cooperating processes is called Inter-process Communication (IPC)

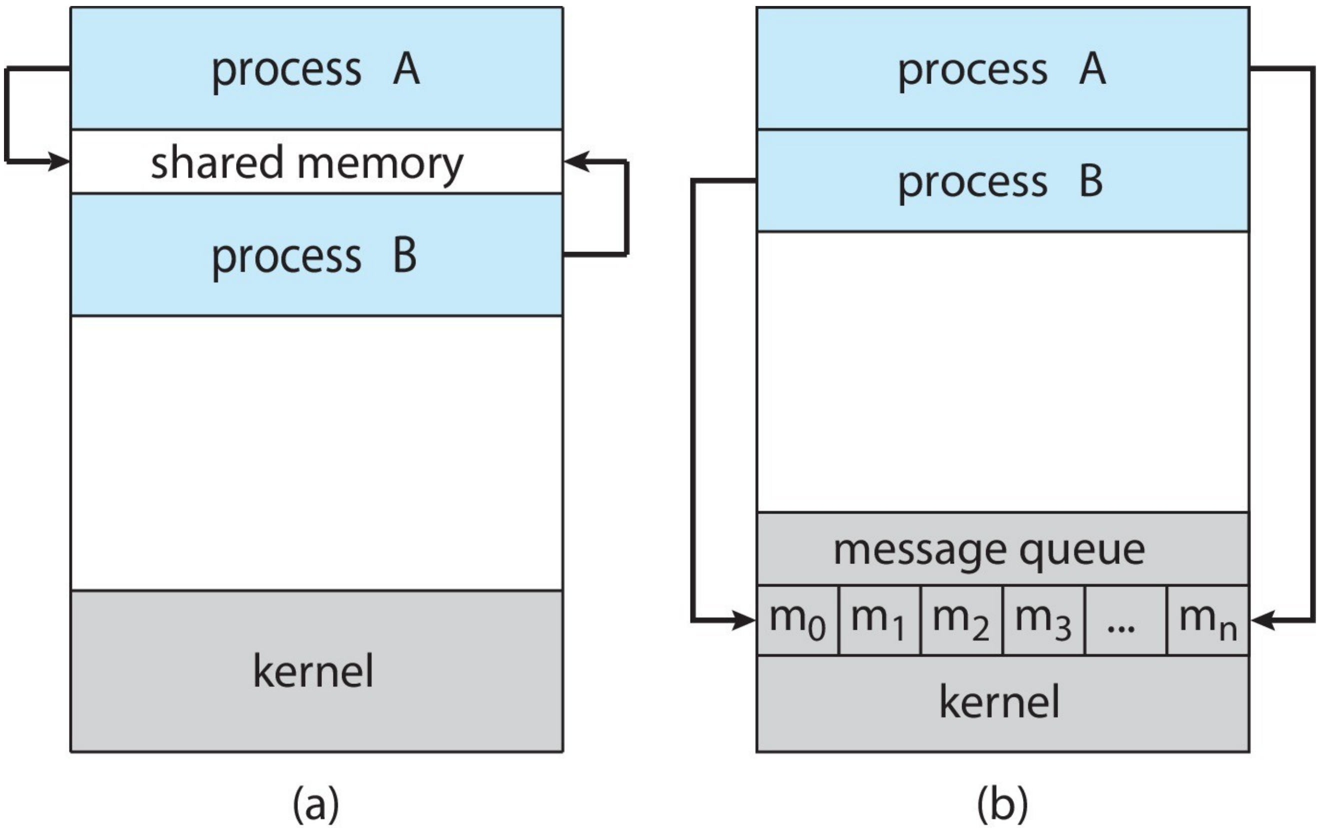
Inter-process Communication (IPC)

- The means of communication for cooperating processes is called Inter-process Communication (IPC)
- Process is designed for isolation, so IPC is not easy
  - Without overhead
- Models of IPC
  - Signal
  - Shared memory
  - Message passing
  - Pipe
  - Socket
  - IPC
  - …



IPC Communication Models

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



IPC Communication Models

- Most OSes 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

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)

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

Bounded-Buffer – Shared-Memory Solution

- The textbook producer/consumer example

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

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
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;
}
```

```
item next_consumed;

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

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

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 beast
- 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
- In this day and age, shm-type code is used very rarely, 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 later in the semester
  - 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

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Message Passing - 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

Message Passing - 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

Message Passing - Indirect Communication

- 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

Message Passing - 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.

Implementing Message-Passing

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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**

Implementing Message-Passing

- The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox.
- When the consumer invokes receive(), it blocks until a message is available.

```
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 */
}
```

Implementing Message-Passing

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Buffering

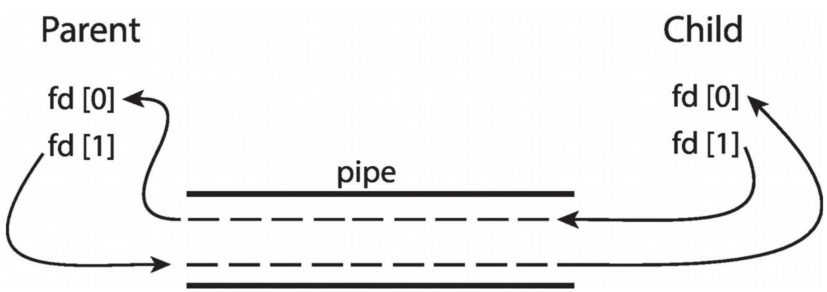
- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

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.

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
  - fd[0] is the read end; fd[1] is the write end



- 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

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 pipes must be used for bi-directional 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

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
  - Java RMI
- Tons of other less used ones (named pipes, shared message queues, etc...)
  - The history of IPCs is huge and the number of IPC implementations/abstractions is staggering

Example: Sockets

- A socket is a communication abstraction with two endpoints so that two processes 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

Remote Procedure Calls

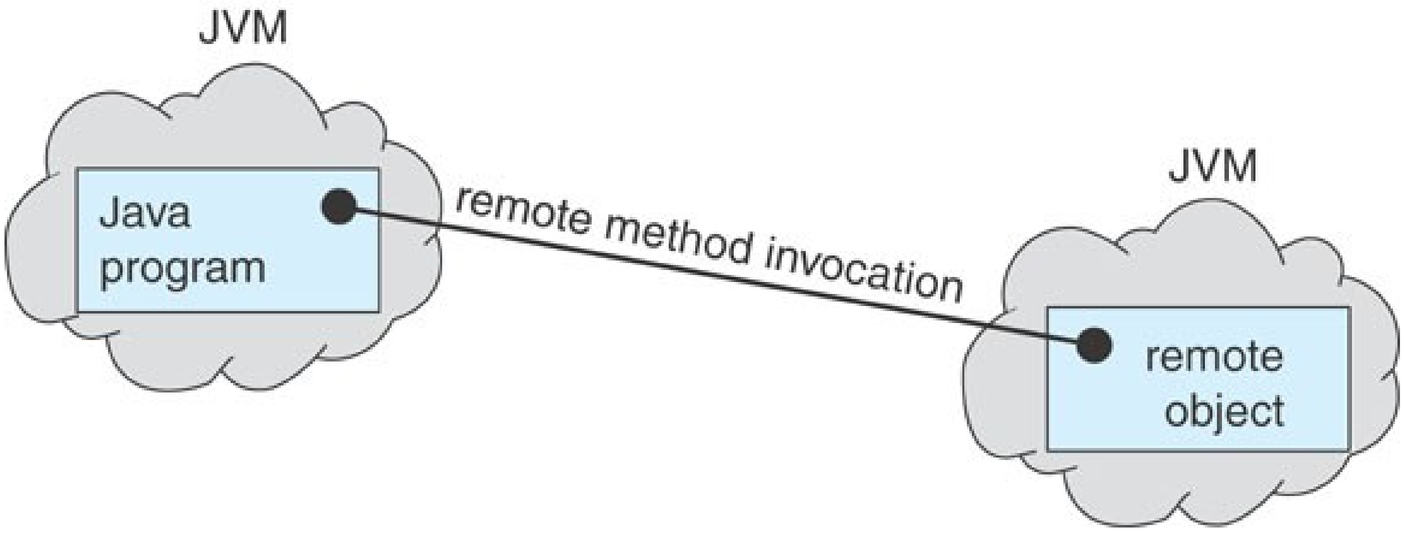
- So far, we've seen unstructured message passing
  - A message is just a sequence of bytes
  - It's the application's responsibility to interpret the meaning of those bytes
- RPC provides a procedure invocation abstraction across hosts
  - A "client" invokes a procedure on a "server", just as it invokes 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

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

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)

Takeaway

- 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, RMI, signals