Interprocess Communication

Reading:

Silberschatz

chapter 4

Additional Reading:

Stallings

chapter 6

Outline

- Introduction
  - Shared memory systems
  - POSIX shared memory
  - Message passing systems
- Direct communication
- Indirect communication
- Buffering
- Exception conditions
- A Case Study for UNIX Signals
  - Using keyboard
  - Using command line
  - Using system calls
- Client-Server communication
  - Sockets
  - Remote procedure calls
  - Remote method invocation

Process Cooperation

- Independent Process
- Cooperating Process
- Why Cooperation?
  - Information Sharing
  - Computation Speed-up
  - Modularity
  - Convenience
- What is IPC?
  - Shared Memory
  - Message Passing

Interprocess Communication

Cooperating process require IPC to exchange data and information. Two models:

- Shared Memory: Read and write data in shared region
  Maximum speed & convenience, within computer

  (*e.g*. UNIX pipes)
- Message Passing: Exchange messages between cooperating process
  Useful for exchanging small amount of data
  Implementation ease for intercomputer communication Vs SM

  (*e.g*. UNIX Sockets)

Shared Memory Systems

Memory speeds, Faster than message passing

- Permission: Normal case, one process cannot access others memory
  - **Shared memory region – *resides* on address space of process creating shared memory region**
  - **Communication process – attach to this address space**
- POSIX Shared Memory:
  - **Process creates shared memory segment**
    - `Segment_id = shmget(IPC_PVT, size, S_IRUSR | S_IWUSR)`
    - IPC_PVT – Identifier to shared memory segment
    - size in bytes
    - mode, S_IRUSR/S_IWUSR – Owner R/W
  - **Other process attach it their address apace**
    - `Shared_memory = (char *) shmat (id, NULL, 0)`
    - id – Integer identifier to shared memory segment
    - Pointer location in memory to attach shared memory, NULL lets OS
    - Flag – 0, both read & write in shared region
  - **Usage**
    - `sprintf(shared_memory, “Learning POSIX Shared Memory Usage”)`
    - `shmdt()` – detach, `shmctl()` - remove

Shared Memory Systems

- Producer-Consumer problem: Common paradigm for cooperating process

  e.g. assembly code from compiler – assembler

  html files & images – client web browser

- Shared Memory Solution: Use a buffer in shared memory filled up by consumer, emptied by consumer, ensure *sync*

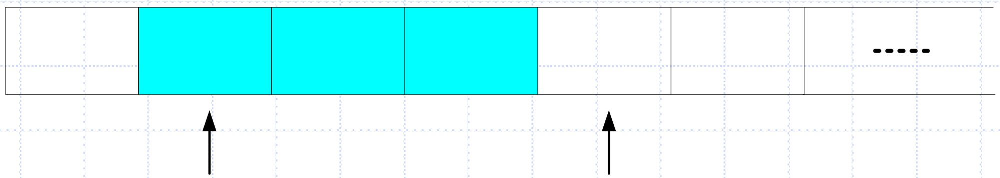
- Unbounded buffer – No limit, producer can always produce, consumer may wait for new items

- Bounded buffer – Fixed buffer size, consumer must wait if empty, producer must wait if full

Producer/Consumer Problem

Several Consumers and producers, Code?

- Buffer hold the *data* which is not yet consumed
- Two logical pointers; **in** and **out**
- **in** (**out**) **-** next (first) free (full) position in the buffer
- **in == out,** Empty; ((**in +1) % BUFFER_SIZE == out**, Full



- Producer Process

```
item nextProduced;

while (1) {
        while ((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer Process

```
item nextConsumed;

while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```


Buffer: 1 2 3 4 5

Message Passing Systems

- Communicate & Sync actions <u>without sharing</u> the same address space
- Useful in *distributed environment*, chat programs on *www*

➢ Fixed size Vs Variable size messages

➢ Basic operations

send (*message*) – transmission of message

receive (*message*) – receipt of a message

Links Logical Implementation rather than its Physical Implementation

➢ Important design issues

- Form of communication – Direct Vs Indirect
- Error handling – How to deal with the exception conditions?
- Buffering – How and where the messages are stored?
  - Automatic or Explicit Buffering

Direct Communication

A communication link in direct communication has following properties

- A link is established automatically between every pair of process wishing to communicate, but the process need to know each others identity

- A unique link is associated with two process

- The link is usually bidirectional but it can be unidirectional

Naming

- Direct Communication

  Process must explicitly name the receiver or sender of a message

- Symmetric Addressing
  - **send** (P, *message*) – Send *message* to process P
  - **receive** (Q, *message*) – Receive *message* from Q
- Asymmetric Addressing

  Variant of above scheme, only sender names the receiver and receiver is does not have to know the name of specific
  - **send** (P, *message*) – Send *message* to process P
  - **receive** (id, *message*) – Receive a pending (posted) *message* from any process, when the message arrives, id is set to the name of process
- Disadvantage – Limited modularity, process identifier

Indirect Communication

Communication Link Properties

- A link is established between two process *only if* they ‘share’ a mailbox
- A link may be associated with more than two process
- Communication process may have different links between them, each corresponding to one mailbox
- The link is <u>usually</u> bidirectional but it can be unidirectional

Indirect Communication

- Messages are sent to *or* received from **mailboxes** or **ports**. The **send** and **receive** primitives can take following forms:
  - **send** (A, *message*) – Send *message* to mailbox A
  - **receive** (A, *message*) – Receive *message* from mailbox A
- This form of communication decouples the sender and receiver, thus allowing greater flexibility
- Generally a mailbox is associated with many senders and receivers
- A mailbox may be owned either by a process or OS
- If mailbox is owned by a process – Owner and user

Synchronization

Design options for implementing **send** and **receive** primitives:

- Synchronous – synchronization required for communicating process
  – both **send** and **receive** are blocking operations
  – also known as *rendezvous*

- Asynchronous – **send** operation is always almost non-blocking
  – **receive** operation can have blocking (waiting) or non blocking (polling) variants

Buffering

Messages exchanged by communicating process resides in a temporary queue. Such queues can be implemented in three ways:

- Zero Capacity
  - No messages waiting, used in synchronous communication
- Bounded Capacity
  - When buffer is full, *sender* must wait
- Indefinite Capacity
  - The *sender* never waits

In non-zero capacity cases (asynchronous) the *sender* is unaware of the status of the message it sends. Hence additional mechanisms are needed to <u>ensure</u> the delivery and receipt of a message.

Exception Conditions

Single machine environment - usually shared memory messages
Distributed environment – messages are occasionally lost, duplicated, delayed, or delivered out of order. Some common exception/error conditions that require proper handling.

- Process Terminates
  - Either a sender or a receiver may terminate *before* a message is processed
- Lost Messages
  - A message may be lost in the communication link sue to hardware/line failure
- Scrambled Messages
  - A message arrives in a state that cannot be processed

- Primitives not suitable for synchronization in distributed systems
  - *Semaphores* require global memory
  - *Monitors* require centralized control

Message passing is a mechanism suitable not only for IPC, but also for synchronization, in both centralized and distributed environments.

A case study – UNIX signals

A UNIX *signal* is a form of IPC used to notify a process of an event.
- **generated** when event first occurs
- **delivered** when the process takes an action on that signal
- **pending** when generated but not yet delivered.

Signals, also called *software interrupts*, generally occur asynchronously

- Signals
  - Various notifications sent to a process to notify it of *important event*
  - They interrupt whatever the process is doing at that time
  - Unique integer number and symbolic name (/usr/include/signal.h)
  - See the list of signals supported in your system <kill -l>
  - Each signal may have a *signal handler*, function that gets called when process receives the signal
- Handling Signals
  - Used by OS to notify the processes that some event has occurred
  - Event notification mechanism for a specific application
- Sending Signals
  - One process to another, including itself
  - Kernel (OS) to process

A case study – UNIX signals

- Sending Signals Using Keyboard
  - Ctrl-C
    - System sends an INT signal (SIGINT) to running process
    - By default – Immediately terminates the running process
  - Ctrl-Z
    - System sends an TSTP signal (SIGSTP) to running process
    - By default – Suspends the execution of running process
- Sending Signals Using Command Line
  - kill - <signal> <PID>
    - Signal name or number, e.g. kill – INT 1560, similar to Ctrl-C
    - If no Signal name?
  - fg
    - Resume the execution of process suspended by Ctrl-Z by sending CONT signal
  - raise <signal>
    - Process sends signal to itself
  - signal <*signal*, SIGARG func>
    - System Call, A process may declare a function to serve a particular signal as above. When *signal* is received,
    - Process is interrupted and func is called immediately, resumes once executed

What to do with a *signal?*

Using the **signal()** system call, a process can:

- Ignore the signal – only two signals, SIGKILL (kill-9 PID) and SIGSTOP (Ctrl-Z) cannot be ignored
- Catch the signal – tell the kernel to call a function whenever the signal occurs
- Let the default action apply – depending upon the signal, the default action can be:

  exit – perform all activities as if the exit system call is requested

  core – first produce core image on the disk and then perform the exit activities

  stop – suspend the process

  ignore – disregard the signal

A case study – UNIX signals

➢ Sending Signals Using System Calls

```
#include <unistd.h>          /* standard UNIX functions, like getpid()*/
#include <sys/types.h>       /* various type definitions, like pid_t */
#include <signal.h>          /* signal name macros, and the kill() prototype */


/* first, find my own process ID */
pid_t my_pid = getpid();     /* now that i got my PID, send myself the STOP signal. */
kill(my_pid, SIGSTOP);
```

A case study – UNIX signals

- Using signal() system call

```
#include <unistd.h>          /* standard UNIX functions, like getpid()*/
#include <sys/types.h>       /* various type definitions, like pid_t */
#include <signal.h>          /* signal name macros, and the kill() prototype */


 /* first, here is the signal handler */
void catch_int(int sig_num)
{
          /* re-set the signal handler again to catch_int, for next time */
         signal(SIGINT, catch_int);
         /* and print the message */
          printf("Don't do that");
          fflush(stdout);
}
 .
 .        /* and somewhere later in the code.... */
 .       /* set the INT (Ctrl-C) signal handler to 'catch_int' */
 signal(SIGINT, catch_int);      /* now, lets get into an infinite loop of doing nothing. */
 for ( ;; )
         pause();
```

A case study – UNIX signals

➢ Core dump

- A **core dump** is an unstructured record of the contents of working memory at a specific time
- Generally used to debug a program that has terminated abnormally (crashed)
- Nowadays, it typically refers to a file containing the memory image of a particular process, but originally it was a printout of the entire contents of working memory
- The name comes from core memory and the image of dumping a bulk commodity (such as gravel or wheat)

➢ Generating Core dump of a running process

- To generate a core file named '*core*' in the current working directory for the process with a process id of 1230, use:
  `<gcore 1230>`

A case study – UNIX signals

- Some possible signals, their #, and their default handling

SIGNAL	ID	DEFAULT	DESCRIPTION
SIGHUP	1	Termin.	Hang up; sent to process when kernel assumes that user of a process is not doing any useful work
SIGINT	2	Termin.	Interrupt. Generated when we enter CNRTL-C
SIGQUIT	3	Core	Generated when at terminal we enter CNRTL-\
SIGILL	4	Core	Generated when we executed an illegal instruction
SIGTRAP	5	Core	Trace trap; triggers the execution of code for process tracking
SIGABRT	6	Core	Generated by the abort function
SIGFPE	8	Core	Floating Point error
SIGKILL	9	Termin.	Termination (can't catch, block, ignore)
SIGBUS	10	Core	Generated in case of hardware fault
SIGSEGV	11	Core	Generated in case of illegal address
SIGSYS	12	Core	Generated when we use a bad argument in a system service call
SIGPIPE	13	Termin.	Generated when writing to a pipe or a socket while no process is reading at other end
……….	…..	………	………
……….	…..	………	………

Many more…

Client-Server Communication

Shared memory, message passing

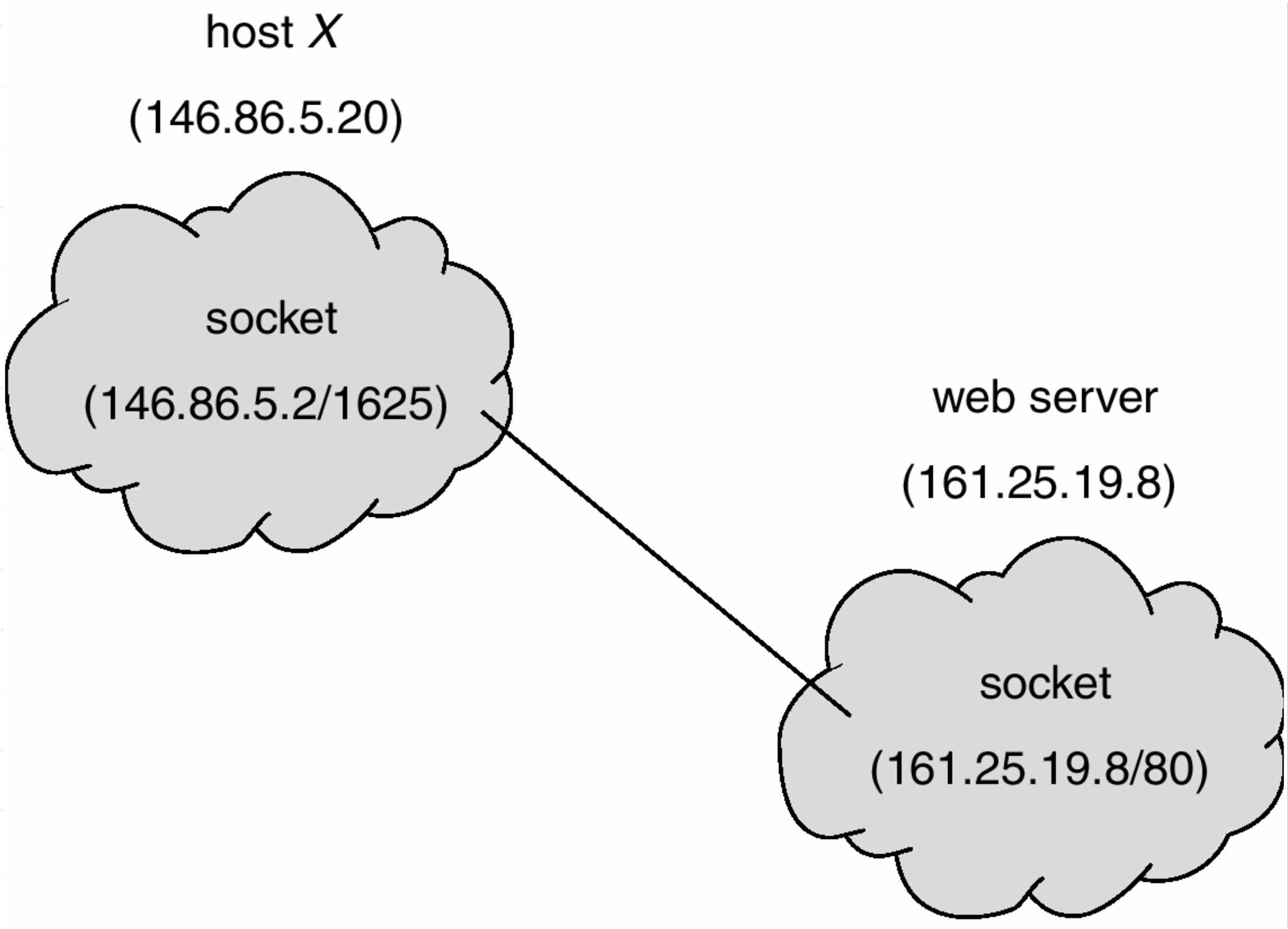
Several other strategies for communication in client-server systems:

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- A socket is identified by an IP address Concatenated with port number
- Sockets use client-server architecture
- 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



Socket Communication - Summary

- Server Side
  1. `socket();`
  2. `bind();`
  3. `listen();`
  4. `accept();`
  5. `send()/recv()`

- Client Side
  1. `socket();`
  2. `connect();`
  3. `send()/recv()`

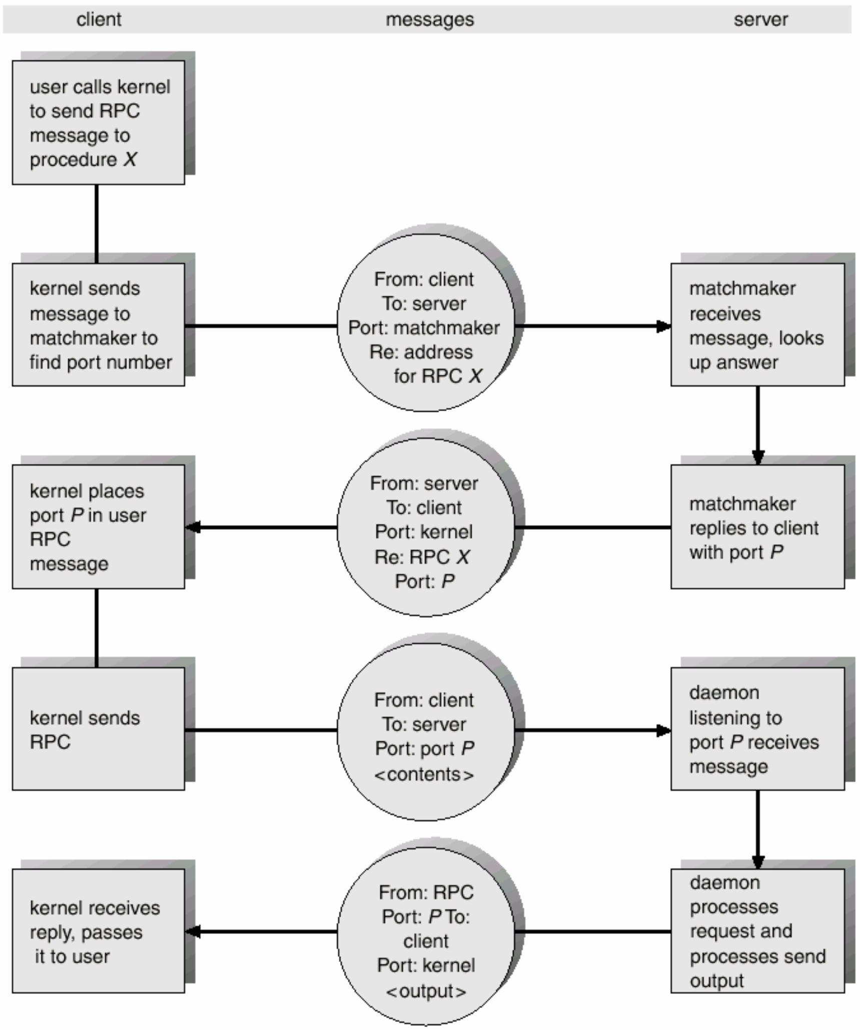
- When you first create the socket descriptor with `socket()`, the kernel sets it to blocking. If you don't want a socket to be blocking, you have to make a call to `fcntl()`:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/socket.h>
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

Remote Procedure Calls

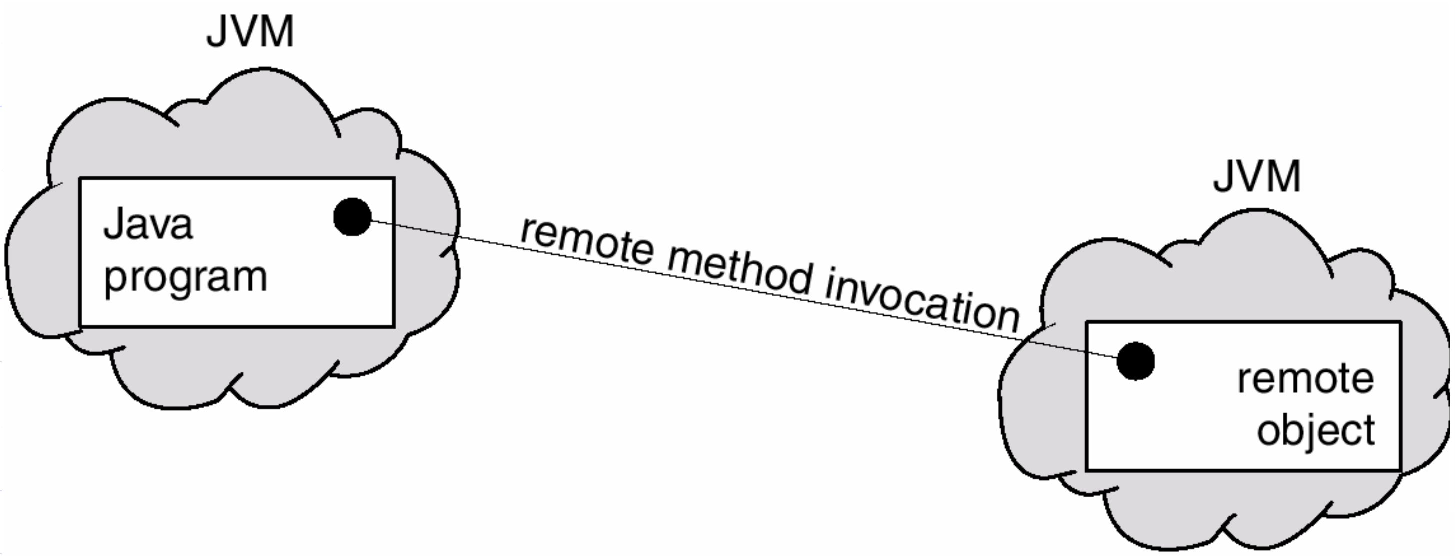
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** $\rightarrow$ Client-side proxy for the actual procedure on the server
  - Client-side stub locates server and *marshalls* the parameters
  - Server-side stub receives this message, unpacks the *marshalled* parameters, and performs the procedure on server
- Data Representation client and server machines
  - *Big-endian* Vs *Little-endian*, XDR
- Semantics
  - *at most once*, *exactly once*

Execution of RPC



Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
  - *RMI = RPC + Object-Orientation*
- RMI allows a Java program on one machine to invoke a method on a remote object



RMI and RPCs

Fundamental differences

- RPCs support *procedural* programming while RMI is object based, it supports invocation of methods on remote objects

- RPCs the parameters to remote procedures are *ordinary data structures*, while it is possible to pass *objects* as parameters to remote procedures (Java applications distributed across the network)

Marshalling Parameters