Section 5.3
Inter-Process Communication

1. Overview
2. Signals
3. Sockets

5.3.1 Overview

- What is inter-process communication (IPC)?
  - it's the sending and receiving of information between processes
    - on the *same* physical host, or
    - on *separate* physical hosts
      - the hosts must be networked together
- Main approaches to IPC
  - signals
  - sockets

5.3.2 Signals

- What is a signal?
  - it's an *integer value* sent from one process to another
    - there is a fixed set of existing signal values
      - **`/usr/include/.../bits/signum.h`**
    - only two of these are *user-defined*
    - signals can be sent from the shell command line too!
  - signals are typically used in *error situations*
    - to tell a process to terminate
  - they are a *very limited* kind of IPC
    - processes must be on the **same host**
    - we **cannot** send signals between two different hosts
    - only predefined signal values can be sent

Signals (cont.)

- There are two steps in using signals in our program:

  1. install a *signal handler* function
     - this tells the process which function to call when a signal is received
     - signal handler must take one **`int`** parameter, and return **`void`**

  2. send a signal
     - send a specific, valid signal from one process to another

- **`coding example <p1>`**

Installing a Signal Handler

- What is a *signal handler*?
  - a **function** to be called when a specific signal is received
- Characteristics
  - every signal has its own handler
  - there is a *default* handler for every signal
    - the default behaviour is usually to terminate the process
  - the signal handler is installed using the **`signal()`** system call

Installing a Signal Handler (cont.)

`sighandler_t signal(int` *`signum`*`, sighandler_t` *`action`*`)`

- Description
  - this system call installs the signal handler specified in ***`action`*** to handle signal ***`signum`***
  - **`sighandler_t`** is a predefined data type
    - it is used for a function that:
      - takes one **`int`** as parameter
      - returns **`void`**
  - it returns the signal handler previously associated with ***`signum`***

Installing a Signal Handler (cont.)

- Description (cont.)
  - ***signum*** must be one of the predefined signal values
  - ***action*** can have one of the following values:
    - **SIG_IGN**
      - tells the process to ignore the signal and do nothing
    - **SIG_DFL**
      - tells the process to call the default signal handler
    - a signal handler function
      - tells the process to call the specified function
- **coding example <p2>**

Sending a Signal

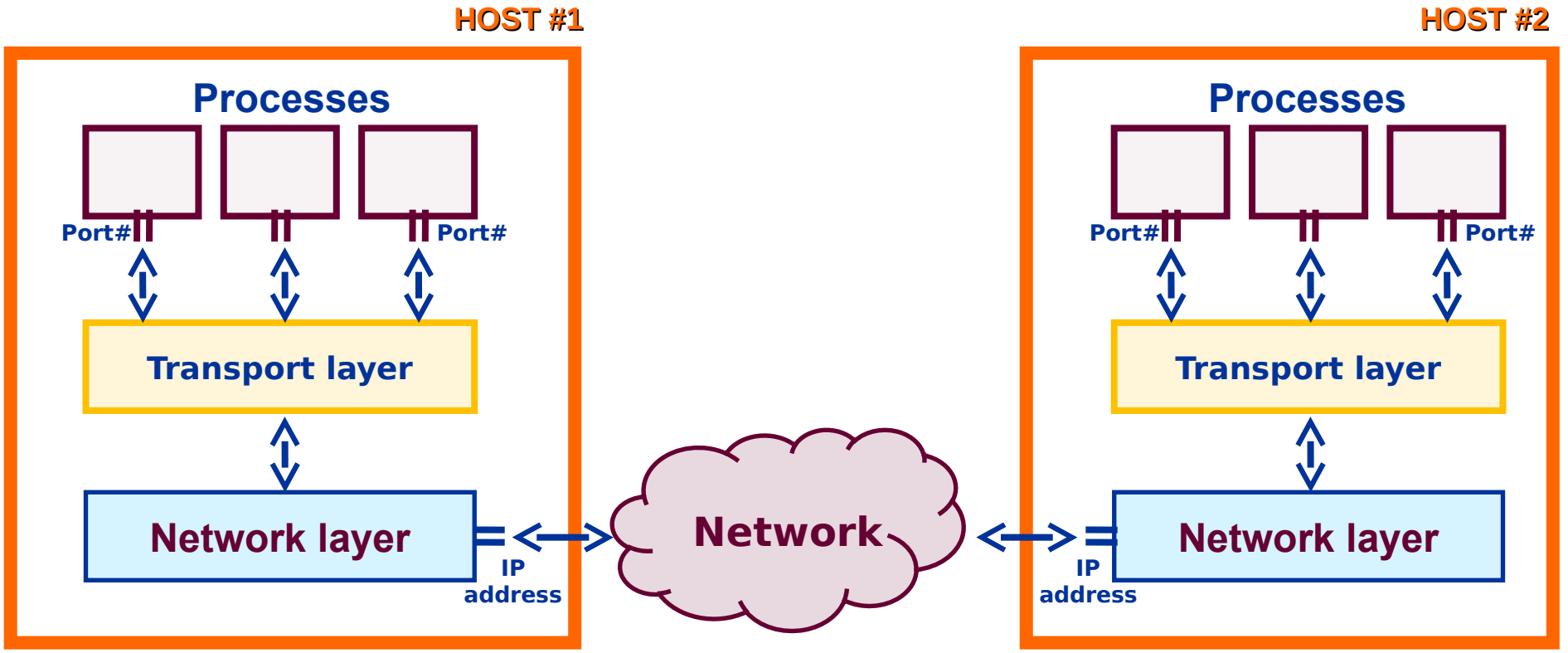
`int kill(pid_t pid, int signum)`

- Description
  - this sends the signal ***`signum`*** to the process with identifier ***`pid`***
  - ***`signum`*** must be one of the predefined signal values
  - return value:
    - zero, if successful
    - -1 in case of error
- **`coding example <p3>`**

5.3.3 Sockets

- What is a *socket*?
  - it's an endpoint in inter-process communication (IPC)
    - the processes can be on the *same* host, or on *different* hosts
  - a socket address is made up of:
    - an Internet Protocol (IP) address
      - this indicates a unique host on a network
    - a port number
      - this indicates a unique application running on that host
      - not the PID! hosts don't know about other hosts' PIDs
  - it is represented as an integer

Basic Networking



- Network layer protocol:
  - Internet Protocol (IP)
- Transport layer protocols:
  - Transmission Control Protocol (TCP)
  - User Datagram Protocol (UDP)

Socket Components

- IP address
  - this uniquely identifies a computer at the network layer
- Port number
  - this uniquely identifies an application at the transport layer
    - remote hosts don’t know an application’s PID
  - only specified range of values is unreserved and can be used

Types of Sockets

- Stream sockets
  - these are *connection-based* sockets
    - connection must be established between sender and receiver first
    - after a connection is established, two-way communication can begin
    - the connection must be closed when communication is finished
  - they are used for:
    - reliable packet delivery
    - packet correctness
    - reliable order of packets
  - they work with the TCP transport protocol

Types of Sockets (cont.)

- Datagram sockets
  - these are *connection-less* sockets
  - they are used for:
    - faster packet delivery
  - they work with the UDP transport protocol
- Raw sockets
  - these bypass transport layer entirely

Socket Communications

- Steps in socket communications
  - each endpoint opens a socket
  - for stream sockets, a connection is established
  - packets are sent and received
  - each endpoint closes its socket
- Socket communications must be *synchronized*
  - one endpoint must be ready to receive if the other is ready to send
  - if the two endpoints get out of sync, a *deadlock* may result

Client-Server Model

- What is the client-server model?
  - it's a type of *system architecture*
    - it's one approach for organizing code in large applications
  - it's a type of IPC architecture
- Characteristics
  - one server process receives requests and performs work
  - one or more client processes send requests to server

Client-Server Model (cont.)

- Steps in establishing connection-based communications
  - server
    - create a stream socket on which to receive connection requests
    - bind the *listening socket* to server's IP address and port number
    - listen on socket for incoming connection request from client
    - accept a connection from client
    - receive and send data on *client socket*
    - close the client socket
    - wait for a new incoming connection request on the listening socket
  - client
    - create a stream socket to connect to the server
    - connect to the server at its IP address and port number
    - send and receive data
    - close the socket
  - **`coding example <p4>`**

Client-Server Model (cont.)

- Steps in establishing connection-less communications
  - server
    - create a datagram socket on which to receive messages
    - bind the socket to server's IP address and port number
    - select incoming message from client
    - receive and send data
    - close the socket
  - client
    - create a datagram socket to connect to the server
    - connect to the server at its IP address and port number
    - send and receive data
    - close the socket
  - **coding example <p5>**