

## Chapter 3: Processes



Operating System Concepts – 10<sup>th</sup> Edition

Silberschatz, Galvin and Gagne ©2018

## Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Operating System Concepts – 10<sup>th</sup> Edition

3.2

Silberschatz, Galvin and Gagne ©2018

## Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Describe client-server communication using sockets and remote procedure calls.

Operating System Concepts – 10<sup>th</sup> Edition

3.3

Silberschatz, Galvin and Gagne ©2018

## Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Operating System Concepts – 10<sup>th</sup> Edition

3.4

Silberschatz, Galvin and Gagne ©2018

## Process Concept (Cont.)

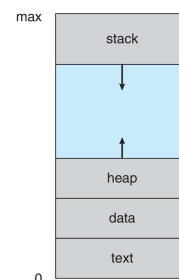
- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Operating System Concepts – 10<sup>th</sup> Edition

3.5

Silberschatz, Galvin and Gagne ©2018

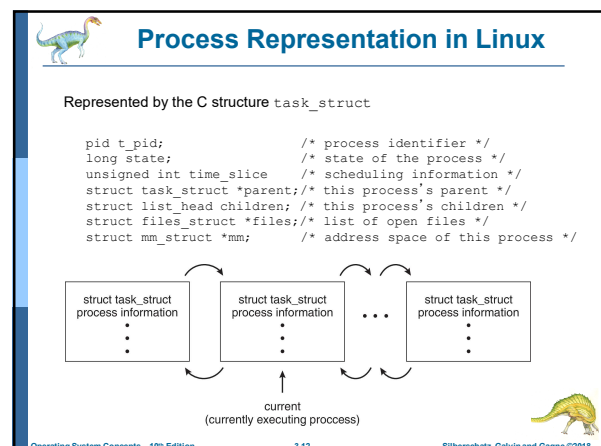
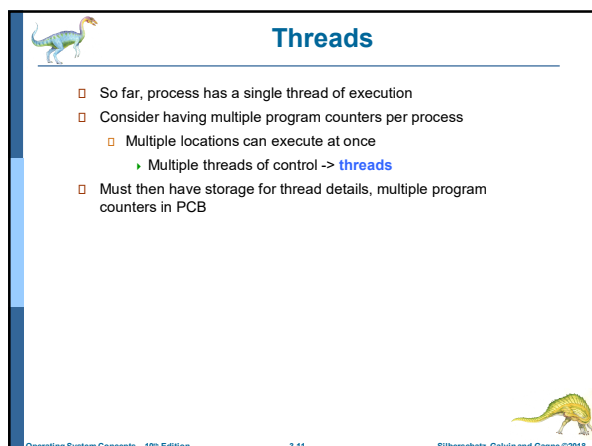
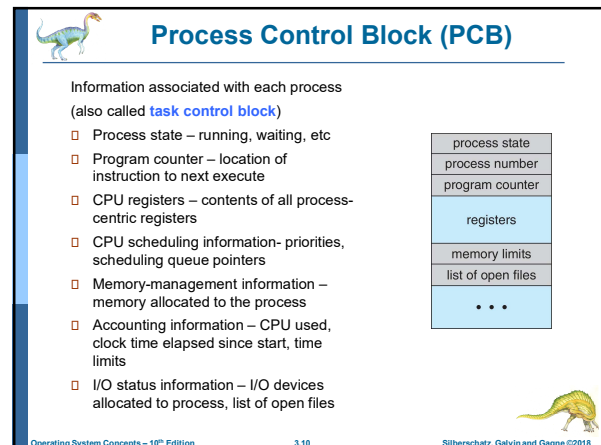
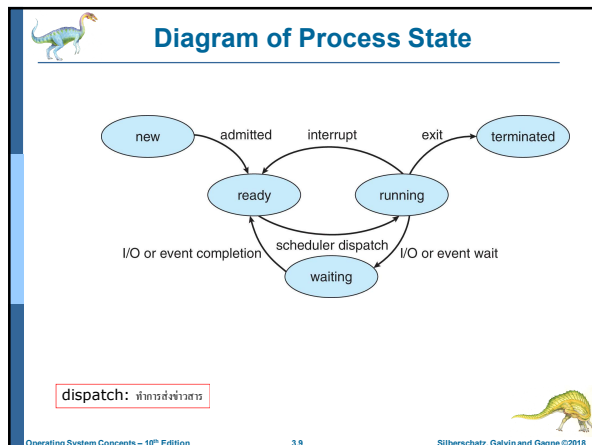
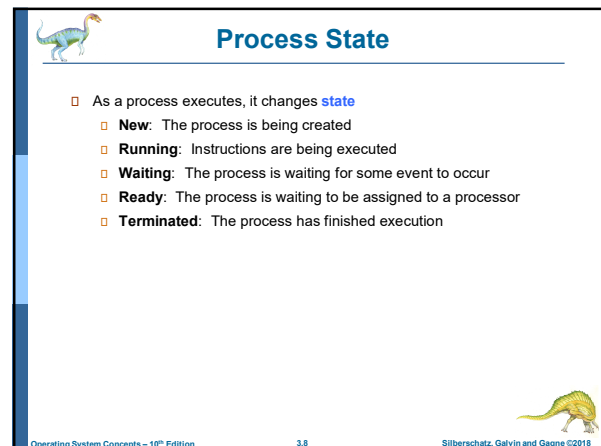
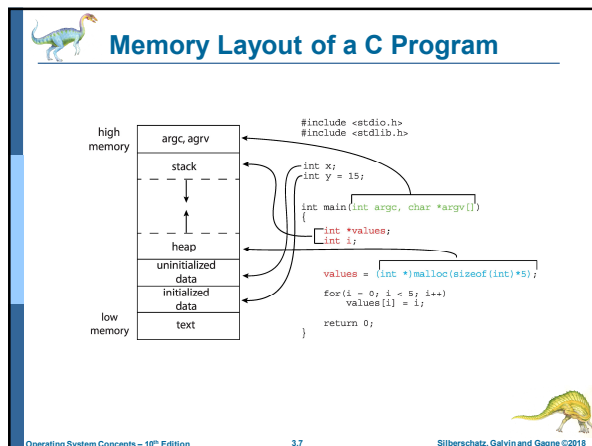
## Process in Memory



Operating System Concepts – 10<sup>th</sup> Edition

3.6

Silberschatz, Galvin and Gagne ©2018



## Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core
- Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
  - Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - Wait queues** – set of processes waiting for an event (i.e. I/O)
  - Processes migrate among the various queues

## Ready and Wait Queues

## Representation of Process Scheduling

Queueing diagram

Each regular box represents a queue

## CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

## Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

## Multitasking in Mobile Systems

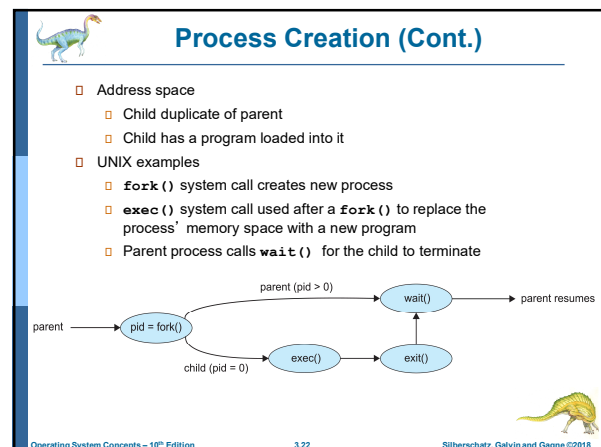
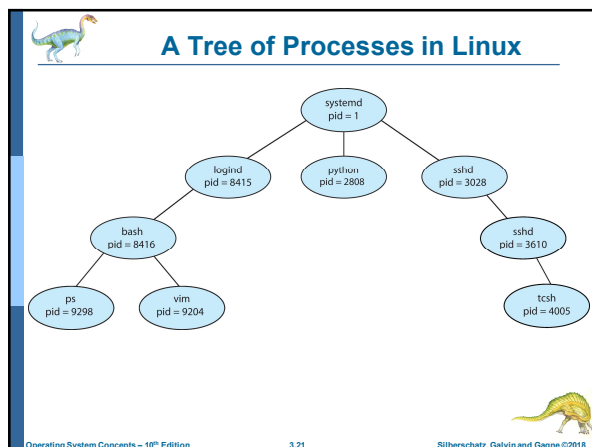
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes- in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

## Operations on Processes

- System must provide mechanisms for:
  - process creation
  - process termination

## Process Creation

- Parent process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



## C Program Forking Separate Process

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        printf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
    
```

## Creating a Separate Process via Windows API

```

#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\cmd.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        printf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
    
```

## Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Operating System Concepts – 10th Edition 3.25 Silberschatz, Galvin and Gagne ©2018

## Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process
 

```
pid = wait(&status);
```

  - If no parent waiting (did not invoke `wait()`) process is a **zombie**
  - If parent terminated without invoking `wait`, process is an **orphan**

Operating System Concepts – 10th Edition 3.26 Silberschatz, Galvin and Gagne ©2018


## Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most to least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.

Operating System Concepts – 10th Edition 3.27 Silberschatz, Galvin and Gagne ©2018

## Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser** process manages user interface, disk and network I/O
  - Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in** process for each type of plug-in



Each tab represents a separate process.

Operating System Concepts – 10th Edition 3.28 Silberschatz, Galvin and Gagne ©2018

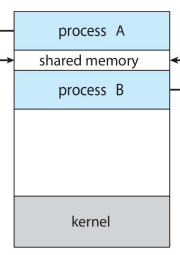
## Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory**
  - Message passing**

Operating System Concepts – 10th Edition 3.29 Silberschatz, Galvin and Gagne ©2018

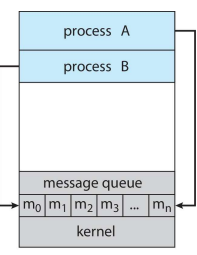
## Communications Models

(a) Shared memory.



(a)

(b) Message passing.




(b)

Operating System Concepts – 10th Edition 3.30 Silberschatz, Galvin and Gagne ©2018

## Cooperating Processes


- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience



Operating System Concepts – 10th Edition 3.31 Silberschatz, Galvin and Gagne ©2018

## Producer-Consumer Problem


- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size



Operating System Concepts – 10th Edition 3.32 Silberschatz, Galvin and Gagne ©2018

## Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.



Operating System Concepts – 10th Edition 3.33 Silberschatz, Galvin and Gagne ©2018

## Bounded-Buffer – Shared-Memory Solution

□ Shared data

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


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

0	1	2	3	4	5	6	7	8	9

buffer

□ Solution is correct, but can only use **BUFFER\_SIZE-1** elements

in : point the next free position in the buffer  
 out: point the first full position in the buffer  
 buffer empty when in==out  
 buffer full when ((in+1) % BUFFER\_SIZE) == out  
 % is modulus operator Ex. 3 % 10 = 3



Operating System Concepts – 10th Edition 3.34 Silberschatz, Galvin and Gagne ©2018

## Producer Process – Shared Memory

```
item next_produced;


while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out); /* do nothing- no free buffers */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

0	1	2	3	4	5	6	7	8	9

buffer

in : point the next free position in the buffer  
 out: point the first full position in the buffer  
 buffer empty when in==out  
 buffer full when ((in+1) % BUFFER\_SIZE) == out  
 % is modulus operator Ex. 3 % 10 = 3



Operating System Concepts – 10th Edition 3.35 Silberschatz, Galvin and Gagne ©2018

## Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
    while (in == out); /* do nothing -nothing to consume*/


    /* remove an item from the buffer */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

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


0	1	2	3	4	5	6	7	8	9

buffer

in : point the next free position in the buffer  
 out: point the first full position in the buffer  
 buffer empty when in==out




Operating System Concepts – 10th Edition 3.36 Silberschatz, Galvin and Gagne ©2018




## Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)`
  - `receive(message)`
- The *message* size is either fixed or variable




Operating System Concepts – 10<sup>th</sup> Edition 3.37 Silberschatz, Galvin and Gagne ©2018




## Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via *send/receive*
- Implementation issues:
  - 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?




Operating System Concepts – 10<sup>th</sup> Edition 3.38 Silberschatz, Galvin and Gagne ©2018




## Message Passing (Cont.)

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




Operating System Concepts – 10<sup>th</sup> Edition 3.39 Silberschatz, Galvin and Gagne ©2018




## 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




Operating System Concepts – 10<sup>th</sup> Edition 3.40 Silberschatz, Galvin and Gagne ©2018




## 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




Operating System Concepts – 10<sup>th</sup> Edition 3.41 Silberschatz, Galvin and Gagne ©2018



## 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*



Operating System Concepts – 10<sup>th</sup> Edition 3.42 Silberschatz, Galvin and Gagne ©2018

## 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.

Operating System Concepts – 10th Edition 3.43 Silberschatz, Galvin and Gagne ©2018

## 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**

rendezvous : จุดนัดพบ

Operating System Concepts – 10th Edition 3.44 Silberschatz, Galvin and Gagne ©2018

## Producer – Shared Memory

```

message next_produced;

while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
    
```

Operating System Concepts – 10th Edition 3.45 Silberschatz, Galvin and Gagne ©2018

## Consumer– Shared Memory

```

message next_consumed;

while (true) {
    receive(next_consumed)

    /* consume the item in next_consumed */
}
    
```

Operating System Concepts – 10th Edition 3.46 Silberschatz, Galvin and Gagne ©2018

## Buffering

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

rendezvous : จุดนัดพบ  
finite: จำกัด  
infinite: ไม่จำกัด

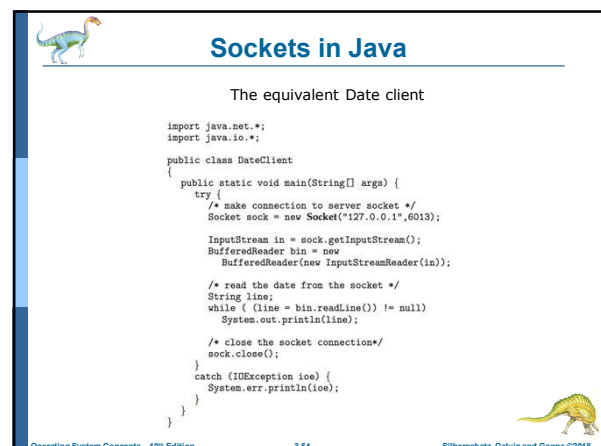
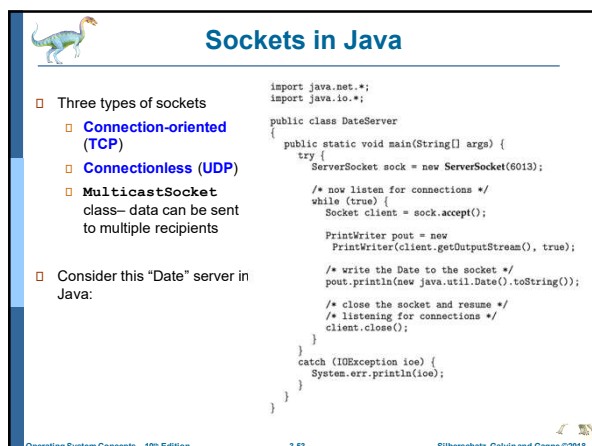
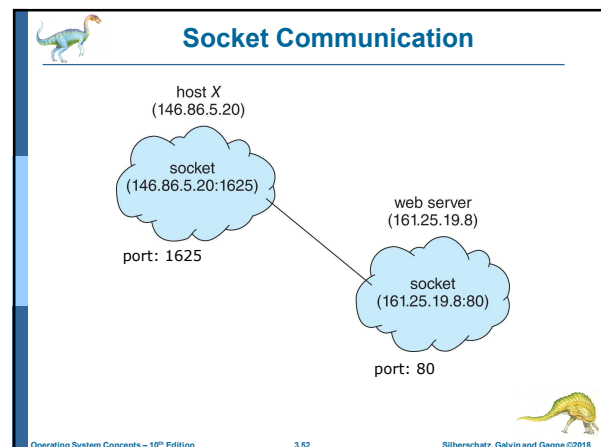
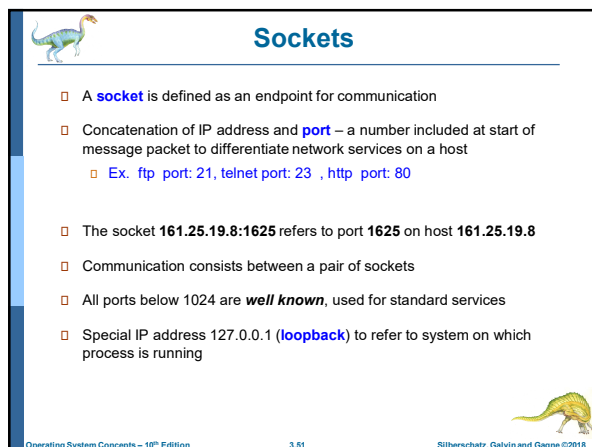
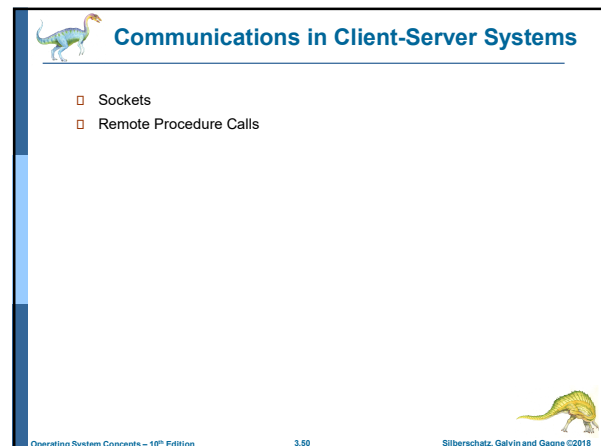
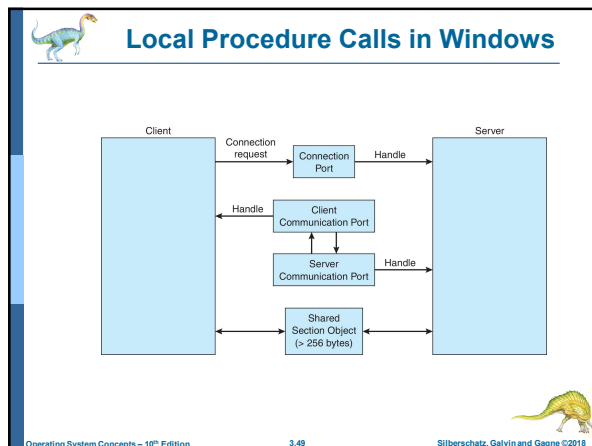
Operating System Concepts – 10th Edition 3.47 Silberschatz, Galvin and Gagne ©2018

## Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (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.


Operating System Concepts – 10th Edition 3.48 Silberschatz, Galvin and Gagne ©2018



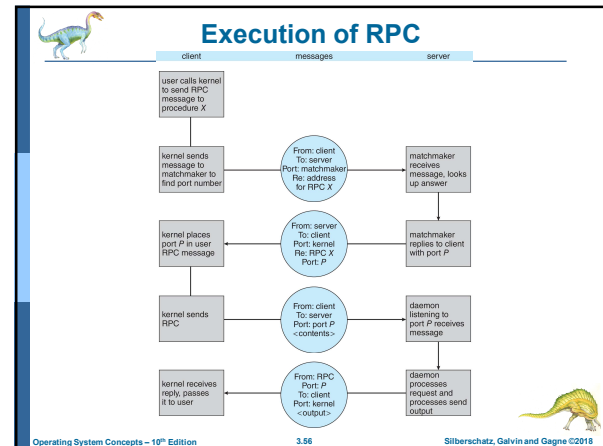


## Remote Procedure Calls


- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (IDL)**



Operating System Concepts – 10th Edition 3.55 Silberschatz, Galvin and Gagne ©2018



## End of Chapter 3



Operating System Concepts – 10th Edition Silberschatz, Galvin and Gagne ©2018