

Daniel **Fitz**  
(43961229)



University Of Queensland  
**COMP3301** – Operating Systems Architecture

ENGG2800 Lecture Notes



# Table of Contents

---

<b>Course Outline</b>	<b>3</b>
<b>Introduction</b>	<b>3</b>
What is an operating system?	3
What Operating Systems Do	3
Operating System Definition	3
Common Functions of Interrupts	4
Interrupt Handling	4
Computer-System Architecture	4
Operating System Structure	4
Operating-System Operations	4
Transition from User to Kernel Mode	5
Process Management	5
Memory Management	5
Storage Management	5
IO Subsystem	5
Protection and Security	6
Open-Source Operating Systems	6
<b>Operating-System Structures</b>	<b>6</b>
Operating System Services	6
User Operating System Interface	6
CLI	7
GUI	7
Touchscreen	7
System Calls	7
System Call Parameter Passing	7
Types of System Calls	7
System Programs	8
Operating System Design and Implementation	9
Implementation	9
UNIX	9
Layered Approach	9
Microkernel System Structure	10
Modules	10
System Boot	10
<b>Processes</b>	<b>10</b>
Process Concept	10
Process State	11
Process Control Block (PCB)	11
Threads	11
Process Scheduling	11

---

Schedulers	12
Addition of Medium Term Scheduling	12
Multitasking in Mobile Systems	12
Context Switch	12
Operations on Processes	12
Process Creation	12
Process Termination	13
Interprocess Communication	13
Cooperating Processes	13
Producer-Consumer Problem	14
Interprocess Communication - Message Passing	14
Direct Communication	14
Indirect Communication	14
Synchronization	15
Buffer	15
Examples of IPC Systems	15
POSIX	15
Communication in Client-Server Systems	15
Sockets	15
Remote Procedure Calls	15
Pipes	16

## List of Tables

---

Placeholder for table of contents	0
-----------------------------------	---

## List of Figures

---

Figure 1: Process in Memory	11
Figure 1: Communications Models	13

---

# Course Outline

---

- Intro
  - What is an OS? Major roles/responsibilities. User/kernel interaction. Operating system structures
- Processes and Threads
  - Operations on processes, Process state, Program vs process, threads vs processes, IPC
- Scheduling
  - Concepts, criteria, algorithms, threads and scheduling
- Deadlock and synchronization
  - Prevention/avoidance/detection/recovery
- Memory management and virtual memory
  - The memory hierarchy, swapping, paging
  - Demand paging, copy-on-write, page replacement, mem-mapped files
- I/O subsystems
  - IO hardware, device models, drivers, interrupt handling, DMA
- Mass storage and filesystems
  - Disks, file systems, mounting, network file systems, disk IO scheduling
- Specialized OSes
  - Real time systems, multimedia systems, embedded systems
- Protection and Security

## Introduction

---

### What is an operating system?

- A program that act as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

### What Operating Systems Do

- Depends on the point of view
- Users want convenience, ease of use
  - Don't care about resource utilization
- But shared computer such as mainframe or minicomputer must keep all users happy
- Users of dedicate systems such as workstations have dedicated resources but frequently use shared resources from servers
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles

### Operating System Definition

- OS is a resource allocator
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is good approximation (But varies wildly)
- "The one program running at all times on the computer" is the kernel. Everything else is either a system program (ships with the operating system) or an application program

---

## Common Functions of Interrupts

- Interrupt transfers control of the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A trap or exception is a software-generated interrupt caused either by an error or a user request
- An operating system is interrupt driven

### Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
  - polling
  - vectored interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

## Computer-System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
  - Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include:
    - 1) Increased throughput
    - 2) Economy of scale
    - 3) Increased reliability – graceful degradation or fault tolerance
- Two types:
  - 1) Asymmetric Multiprocessing
  - 2) Symmetric Multiprocessing

## Operating System Structure

- Multiprogramming needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via job scheduling
  - When it has to wait (for I/O for example), OS switches to another job
- Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing
  - Response time should be <1 second
  - Each user has at least one program executing in memory → process
  - If several jobs ready to run at the same time → CPU scheduling
  - If processes don't fit in memory, swapping moves them in and out to run
  - Virtual memory allows execution of processes not completely in memory

## Operating-System Operations

- Interrupt driven by hardware
- Software error or requests creates exception or trap
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode

- 
- Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
  - Increasingly CPUs support multi-mode operations
    - i.e. virtual machine manager (VMM) mode for guest VMs

## Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

## Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a passive entity, process is an active entity
- Process needs resources to accomplish its task
  - CPU, memory, IO, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

## Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

## Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit – file
  - Each medium is controlled by device (i.e. disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

## IO Subsystem

- 
- One purpose of OS is to hide peculiarities of hardware devices from the user
  - IO subsystem responsible for
    - Memory management of IO including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
    - General device-driver interface
    - Drivers for specific hardware devices

## Protection and Security

- Protection – any mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service

## Open-Source Operating Systems

- Operating system made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement
- Started by Free Software Foundation (FSF), which has "copyleft" GNU Public License (GPL)
- Examples include GNU/Linux and BSD UNIX (including core of Mac OS X), and many more

## Operating-System Structures

---

### Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** – Almost all operating systems have a user interface (UI)
    - Varies between Command-line (CLI), Graphics User Interface (GUI), Batch
  - **Program execution** – The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **IO operations** – A running program may require IO, which may involve a file or an IO device
  - **File-system manipulation** – The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in IO devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** – When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as IO devices) may have general request and release code
  - **Accounting** – To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** – The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external IO devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link

## User Operating System Interface

---

## CLI

CLI or command interpreter allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – shells
- Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

## GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory)
- Many systems now include both CLI and GUI interfaces

## Touchscreen

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry

## System Calls

### System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

### Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management



- 
- request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
  - Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get and set process, file, or device attributes
  - Communications
    - create, delete communication connection
    - send, receive messages if message passing model to host name or process name
      - from client to server
    - shared-memory model create and gain access to memory regions
    - transfer status information
    - attach and detach remote devices
  - Protection
    - control access to resources
    - get and set permissions
    - allow and deny user access

## System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls
- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** – Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info – date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output of the terminal or other output devices
  - Some systems implement a registry – used to store and retrieve configuration information
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** – Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution** – Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** – Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another
- **Background Services**

- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as services, subsystems, daemons
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

## Operating System Design and Implementation

- Design and Implementation of best OS not "solvable", but some approaches have proven successful
- User goals and System goals
  - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Important principle to separate
  - Policy: What will be done?
  - Mechanism: How to do it?
- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- Specifying and designing OS is highly creative task of software engineering

## Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - System programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware (but slower)
- Emulation can allow an OS to run on non-native hardware

## UNIX

- limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

## Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface

- 
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

## Microkernel System Structure

- Moves as much from the kernel into user space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

## Modules

- Most modern operating systems implement loadable kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc

## System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loaded from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running

## Processes

### Process Concept

- An operating system executes a variety of programs
  - Batch system – [jobs](#)

Time-sharing systems – [user programs](#) or [tasks](#)

- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also call [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
- 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

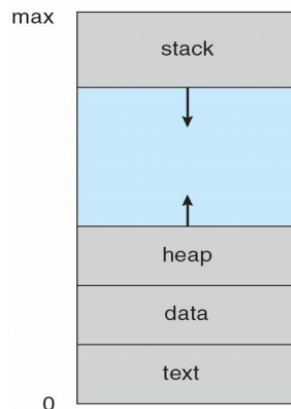


Figure 1: Process in Memory

## Process State

- As a process executes, it changes **state**
  - **New:** The process is being created
  - **Running:** Instructions are being executed
  - **Waiting:** The process is waiting for some event to occur
  - **Ready:** The process is waiting to be assigned to a processor
  - **Terminated:** The process has finished execution

! [Diagram of Process State](sem2-2017/comp3301/state.png)<sub>75</sub>

## Process Control Block (PCB)

Information associated with each process (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

## Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple location can execute at once
    - Multiple threads of control → **threads**
- Must then have storage for thread details, multiple program counters in PCB

## Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device

- 
- Processes migrate among the various queues

## Schedulers

- **Long-term scheduler** (or [job scheduler](#)) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or [CPU scheduler](#)) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
- Short-term scheduler is invoked very frequently (milliseconds) → (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) → (may be slow)
- The long-term scheduler controls the [degree of multiprogramming](#)
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

## Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution → [swapping](#)

## Multitasking in Mobile Systems

- Some systems / early systems allow only one process to run, other 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

## 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 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 → longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

## Operations on Processes

- System must provide mechanisms for process creation, termination, and so on as detailed

## 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
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

## Process Termination

- Process executes last statement and asks the operating system to delete it (`exit()`)
  - Output data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (`abort()`)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated – [cascading termination](#)
- Wait for termination, returning the pid:
 

```
1 | pid t_pid; int status;
2 | pid = wait(&status);
```
- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

## 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**

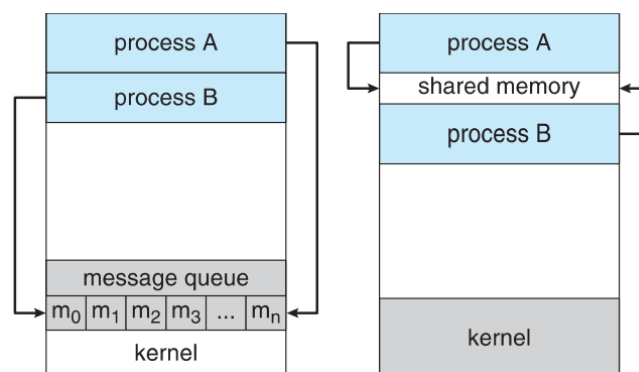


Figure 1: Communications Models

## 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 other process
  - Advantages of process cooperation
    - Information sharing
    - Computation speed-up
    - Modularity
    - Convenience

## Producer-Consumer Problem

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

## 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)** – message size fixed or variable
  - **receive(message)**
- If *P* and *Q* wish to communicate, they need:
  - establish a **communication link** between them
  - exchange message via send/receive
- Implementation of communication link
  - physical (e.g. shared memory, hardware bus)
  - logical (e.g. direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

## 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

## 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
- Operations
  - create a new mailbox
  - 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

## Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**
- Producer-consumer becomes trivial

## Buffer

- Queue of messages attached to the link; implemented in one of three ways
  - 1) Zero capacity – 0 messages  
Sender must wait for receiver (rendezvous)
    - 1) Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
    - 1) Unbounded capacity – infinite length  
Sender never waits

## Examples of IPC Systems

### POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```
  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```
  - Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

## Communication in Client-Server Systems

### Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are *well known*, used for standard services
- Special IP address **127.0.0.1 (loopback)** to refer to system on which process is running

### 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 (MIDL)**
- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered *exactly once* rather than *at most once*
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

## 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

- 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
- 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