COMP 3300
Operating systems fundamentals

Dr. Imran Ahmad

Course Overview

&

Introduction

Course Information

2

- Office:
  - LT 8112
  - Phone : (519) 253-3000 / 3715
  - FAX : (519) 973-7093
  - Email : imran@uwindsor.ca
  - URL : http://cs.uwindsor.ca/~imran
- LECTURE:
  - Tue./Thu. 10:00 – 11:20 AM in TC 104
- OFFICE HOURS:
  - Tue/Thu: 11:20 – 01:00 PM in my office;
  - Wed: 12:50 – 01:20 PM in my office or ER 3107.
  - Any other time, by appointment ONLY.
- GA / TA: Anjali Shah, Jacob Mclean & Kamya Singla

Course Information

3

- Required Text:

  Operating System Concepts (9th Edition)" by A. Silberschatz, P. Galvin & G. Gagne

- Course Blackboard site:
  - Course Outline
  - Important Announcements
  - Homework Assignments
  - Lecture slides
  - Scores
  - Other useful/related links

Course Information (contd.)

4

About the course:

- COMP 3300 is an introduction to the principles underlying the design and implementation of contemporary computer operating systems.
- It will contain a general study of fundamental concepts pertinent to the operating system design and operation and various performance related issues and policies.
- Calendar Description:

  Operating system services, introduction to primary components of multi-programming operating systems, CPU scheduling, concurrent processes, process synchronization and interprocess communication, deadlocks, memory management, file systems, virtual memory, disk scheduling

Learning Objectives:

5

- At the end of the course, the student should have a basic understanding of:
  - Design and implementation issues of contemporary operating systems
  - operation system design such as microkernels, monolithic design, modular design.
  - processes, multithreading, process scheduling, symmetric multiprocessing
  - Memory management techniques, and virtual memory
  - approaches to process synchronization, concurrency and deadlocks
  - Operating system control of Input / Output
  - Operating system management of files

Pre-requisites:

- COMP2120,
- COMP2540,
- COMP2560 and
- COMP2650 OR COMP2660

6

Homework assignments

- Up to 6 individual written and/or programming homework assignments.
  - Due at the beginning of class time on its due date.
  - MUST be type written.
  - Submission will be through Blackboard.
  - NO late submission is allowed.
    - Blackboard timestamp will determine submissions time.
  - More details later.
  - **ALL** assignments **will be** individual assinments
    - no copying from ANY other source (see policy on cheating/copying).
  - For programming assignments, **NO sample solution** will be provided.

7

Exams:

- 2 Midterms.
- Final (comprehensive)– Date/time will be determined by the Registrar Office.
- Missing exam policy (details in course outline)
  - If a midterm exams is missed – it percentage will be carried over to the next midterm or final exam but:
    - **ONLY** for a valid and verifiable reason.
    - Under normal circumstances, **MUST** give a prior notice – before the start of exam through email, phone, fax.
    - **YOUR** responsibility to make sure I have received it.
    - For medical conditions, provide a doctors note, stating medical condition on the provided medical form and **not patients statement**.
    - **No notice – no percentage carried over or a makeup exam**.
  - If final exam is missed (due to a valid reason), a (cumulative) makeup exam on **Apr. 30, 2019 @ 11:00 AM**.

Policy on cheating:

- Expected to do all work on individual basis
- For cases of suspected cheating – senate bylaws will be followed and names of involved will be forwarded to administration for necessary actions.
- No negotiations with those involved.
- For more details, see the course outline.

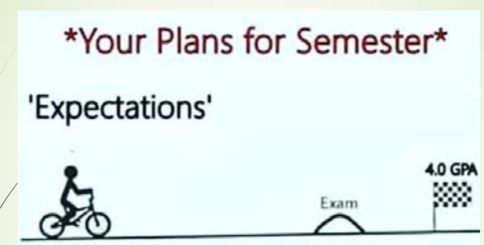
Evaluation scheme:

- **Homework Assignments (up to 6):** 10%
- **Spot Quizzes:** 5%: 3 – 5; min points for each will be 1 %
- **2 Midterms:**
  - Midterm I: 15% (tentatively: Sat. Feb. 09 @ 10:00 AM)
  - Midterm 2: 25% (tentatively: Sat. Mar. 16 @ 10:00 AM)
- **Final (comprehensive):** 45%

Important dates:

- Jan. 16, 2019: Last add/drop day (full tuition refund).
- Feb. 16–24, 2019: Family Day & Study Week – no classes
- Mar. 13, 2019: Last day for voluntary withdrawal
- Apr. 03, 2019: Last day of classes.

Semester Expectations:

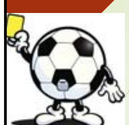


Reality:

Operating Systems

- Important Questions:
  1. What is an operating system?
  2. What does it do?

What is an operating system?



- Referee
  - Manages sharing of resources, Protection, Isolation

- Illusionist
  - Provide clean, easy to use abstractions of physical resources.
    - Infinite memory, dedicated machine
    - Higher level objects: files, users, messages
    - Masking limitations

- Glue
  - Common Services
    - Storage, Interface, Networking
    - Sharing, Authorization
    - Look and feel

What is an operating system? (contd.)

- No standard definition
  - operating systems arose historically as people needed to solve problems associated with using computers.
- A program that acts as an intermediary between a user of a computer and the computer hardware.
- Software that makes computing power available to users by controlling the hardware.
- A collection of software modules including device drivers, libraries, and access routines.
- “*Everything a vendor ships when you order an operating system*” is also a good approximation.

What is an operating system? (contd.)

- OS is a **resource manager / allocator**
  - Manages all resources

  - Resolve contention over finite resources for efficient and fair use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer
- "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.

Operating Systems

- User View varies according to the interface being used
  - **Single User View**
    - maximize the work (or play) that the user is performing
    - OS is designed mostly for **ease of use**, with some attention to **performance** and almost none to **resource utilization**
  - **Multi-user View**

    - users **share resources** and may exchange information.
    - OS is designed to **maximize** resource utilization.

Operating Systems

- Little or **no user view**.
  - embedded computers in home devices, and automobiles
    - Digital alarm clocks
    - Electronic parking meters and parking pay stations
    - Robotic vacuum cleaners
    - Smart watches and digital wrist watches
    - GPS navigation devices
    - Heart rate monitors and pacemakers
    - Cruise control, Anti-lock braking system (ABS),
    - Entertainment systems, etc.
- **Handheld computing Devices**
  - standalone units for individual users
  - OS are designed mostly for **individual usability**, but performance **per amount of battery life** is important

Operating Systems

- **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 Does a Modern OS Do?

- **Provides Abstractions:**
  - Hardware has low-level physical resources with complicated, idiosyncratic interfaces.
  - OS provides abstractions that present clean interfaces with the goal: **make computer easier to use**.
  - Examples:
    - Multitasking, Unbounded Memory, Files, Synchronization and Communication Mechanisms.
- **Provides Standard Interface:**
  - Goal: portability.
  - Unix runs on many different computer hardware platforms.

What Does a Modern OS Do?

- **Mediates Resource Usage:**
  - **Goal:** allow multiple users to share resources fairly, efficiently, safely and securely.
  - Examples:
    - Multiple processes share one processor (pre-emptable resource)
    - Multiple programs share physical memory (pre-emptable resource).
    - Multiple users and files share one or more storage device (non pre-emptable resource).
    - Multiple programs share a given amount of storage device and network bandwidth (pre-emptable resource).

Present and The Future…

- Computers will continue to become physically smaller and more portable.
- Operating systems have to deal with issues like disconnected operations and mobility.
- Media rich information is already within the grasp of common people – e.g., information with pseudo-real time components like voice and video.
- Operating systems will have to adjust to deliver acceptable performance for these new forms of data.

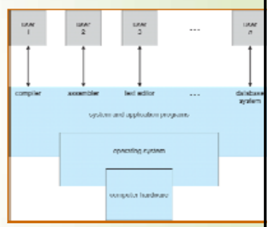
Finally

- Operating systems are so large no one person understands whole system.
  - Have outlived any of its original builders.
- The major problem facing computer science today is how to build large, reliable software systems.
- Operating systems are one of very few examples of existing large software systems, and by studying operating systems we may learn lessons applicable to the construction of larger systems.

Computer System Structure

Computer system can be divided into four components

- **Hardware**
  - Set of basic computing resources CPU, memory, I/O devices
- **Operating system**
  - Controls and coordinates use of hardware among various applications and users
- **Application programs**
  - define ways in which system resources are used to solve computing problems
    - Word processors, compilers, browsers, database systems, video games
- **Users**
  - People, machines, other computers



Operating System Services

- Program development
  - Editors, debuggers, frameworks
- Program execution
  - Initialization, scheduling
- Access to I/O devices
  - Uniform interface, hides details
- Controlled access to files
  - Authorization, sharing, caching

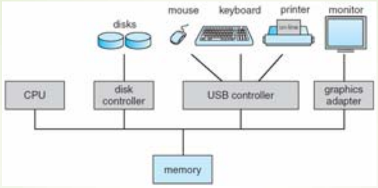
OS Services (continued...)

- System access
  - Protection, authorization, resolve conflicts
- Error detection and response
  - Hardware errors: memory error or device failure
  - Software errors: arithmetic errors, access to forbidden memory locations, allocation errors
- Accounting
  - collect statistics (billing)
  - monitor performance
  - anticipate future enhancements

OS as a resource manager

- OS executes same way as ordinary computer software - it is a set of computer programs. **The key difference is in the intent**
  - Directs use of resources
  - Relinquishes control of the processor to execute other programs
- **Kernel** or nucleus
  - Portion of operating system that is in main memory
  - Contains most-frequently used functions

Computer system operation



- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an *interrupt*.

Common Functions of Interrupts

- An Interrupt
  - transfers control to the interrupt service routine (ISR).
- Requires:
  - to save the address of the interrupted instruction.
  - disable incoming interrupts while processing an interrupt to prevent a *lost interrupt*.
- 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

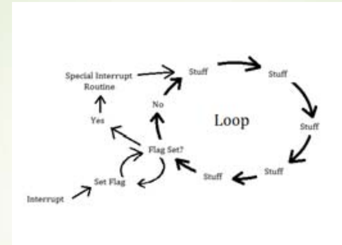
- The OS preserves the state of the CPU by storing registers and the program counter (PC).
- Determines which type of interrupt has occurred.
- Two Methods:
  - *polling*
  - *vectored* interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Handling

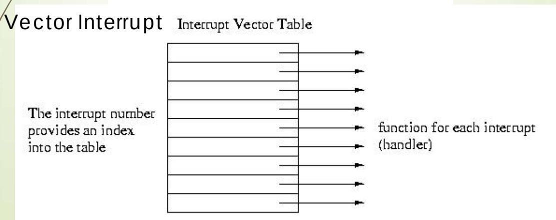
- The OS preserves the state of the CPU by storing registers and the program counter (PC).
- Determines which type of interrupt has occurred.
- Two Methods:
  - *polling*
  - *vectored* interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt system

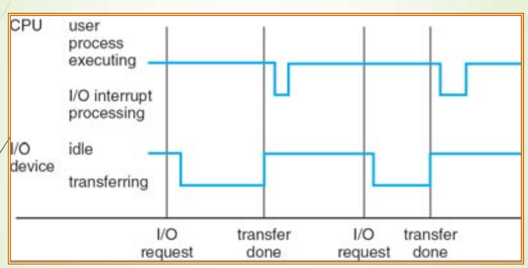
Polling



Vector Interrupt



Interrupt Timeline



I/O Structure

Once I/O is started, two methods:

- **<u>Synchronous I/O</u>**

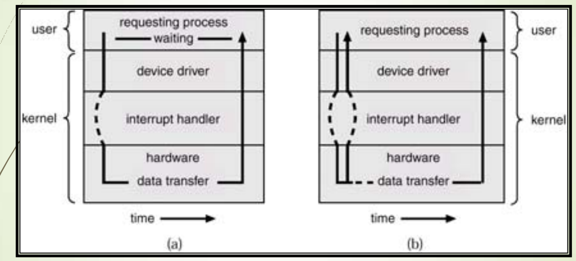
  ➔Control returns to user program <u>only upon I/O completion</u>.

  - How to implement?
  - **Advantage:** At most one outstanding I/O request
  - **Disadvantage:** No simultaneous I/O processing

I/O Structure

- **asynchronous I/O**
  - ➔Control returns to user program <u>without waiting for I/O completion</u>.
  - Needs *System call* – request to the operating system to allow user to wait for I/O completion.
  - Needs to keep track of many I/O requests at same time.
  - *Device-status table* contains entry for each I/O device indicating its
    - Device type, device address, and device state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

I/O Structure



Evolution of Operating Systems

- Operating systems have evolved due to:
  - New types of hardware / hardware upgrades
  - Development of new services and needs
  - Fixes to OS faults
  - Demand / requirements:

Serial Processing

Simple Batch Processing

Multi-programmed Batch Systems

Time-Sharing Systems

Distributed Processing Systems

????

Serial Processing

- Serial Processing
  - No operating system
  - Machines run from a console with display lights and toggle switches, input device, and printer
  - Schedule time
  - Setup included loading the compiler, source program, saving compiled program, and loading and linking

Simple Batch Systems

- Simple Batch Systems
  - Monitors
    - Software that controls running the programs
    - monitor is resident in main memory, ready for execution
    - Batch jobs together, control returns back to monitor
  - Job Control Language (JCL)
    - Special type of programming language
    - Provides instructions to the monitor such as compiler/data to use
  - Hardware Features
    - Memory protection
    - Timer

Operating System Structure

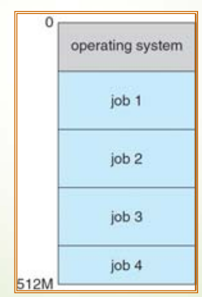
- Single User System
  - designed to manage the computer so that one user can effectively do ONLY one thing at a time.
  - Examples: MSDOS, WINDOWS 3X, WINDOWS 95/97/98 , PalmOS for Palm handheld single-task operating system.
  - Once a process is begun, it must be allowed to run until finished

Operating System Structure

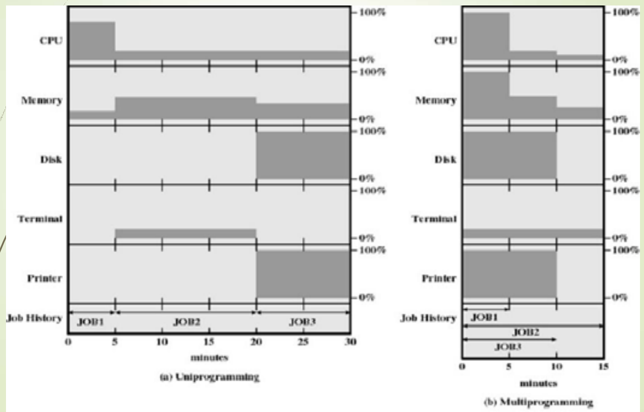
- **Multiprogramming** needed for efficiency
  - Single user - CPU mostly idle whereas I/O devices may be busy
  - organizes jobs (code and data) to keep CPU busy
  - A subset of total jobs in system (*job pool*) is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (e.g., for I/O completion), OS switches to another job
  - As long as there is some job to execute, CPU is not idle.

Memory Layout for Multiprogrammed System

- 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



Multiprogramming



Effects of Multiprogramming

	Uni-programming	Multiprogramming
Processor use	22%	43%
Memory use	30%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min.	15 min.
Throughput rate	6 jobs/hr	12 jobs/hr
Mean response time	18 min.	10 min.

Time-Sharing (Multitasking) Systems

- multiprogramming → effective use of system resources but no user interaction with the system.
- timesharing systems allow several users to interact with system at the same time
- CPU switches jobs very frequently
  - users can interact with each job while it is running, creating **interactive** computing
- Emphasizes **response time** over processor use (< 1 second)

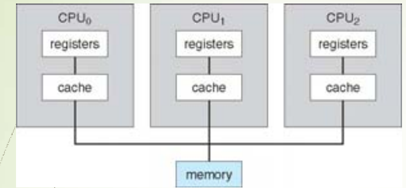
Computer-System Architecture

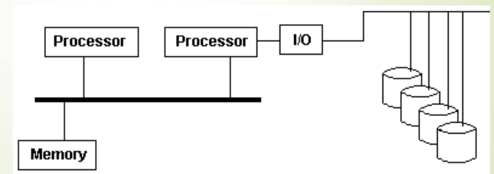
- Most systems use a single general-purpose processor
  - May have special-purpose processors, e.g. GPU
- Multiprocessors
  - Also known as parallel systems OR tightly-coupled systems
  - Advantages include:
    1. Increased throughput
    2. Economy of scale
    3. Increased reliability
       - Graceful Degradation: ability to continue providing service proportional to the level of surviving hardware
       - Fault Tolerance: ability to continue even after failure of a component

Computer-System Architecture

- Two types:
  1. **Symmetric Multiprocessing**
     - All processors are peers
     - Each processor performs the task within OS
     - All processors share memory
     - Example of SMP system is Solaris
  2. **Asymmetric Multiprocessing**
     - Master – slave relationship
     - Each processor is assigned a specific task.
     - Each processor has its own memory
     - A master processor controls the system (other looks for task or have defined tasks)

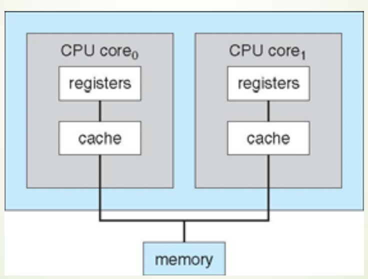
Asymmetric and Symmetric Multiprocessing Architecture





A Dual-Core Design

- **Multicore**
  - Chassis contains multiple separate systems



TERMS TO KNOW AND REMEMBER:

- **Single user system**.
- **Batch systems**
  No timing constraints. To speed up the processing, several similar jobs are put together as a group ➔ better system utilization.
- **Multiprogramming**
  Several programs in memory at same time so that CPU always has something
- **Multiprocessing**
  Several jobs are handled at (virtually) same time.
- **Time-sharing (multitasking)**
  CPU executes multiple jobs by switching among them
- **Interactive Systems**
  Provide direct communication between the user and the system.
- **Multiprocessor System**
  System with >= 1 CPU & all share system bus, clock and memory.
- **Parallel systems**

TERMS TO KNOW AND REMEMBER:

- **Graceful degradation**

  With multiple resources, if a resource fails, work continues with reduced efficiency.
- **Fault tolerant Systems** – systems those support graceful degradation.
- **Real-time systems**

  used when there are rigid time requirements (e.g. space shuttle, control systems,)
- **Networked Systems**

  allows different processes on different systems to share information on network
- **Distributed systems**

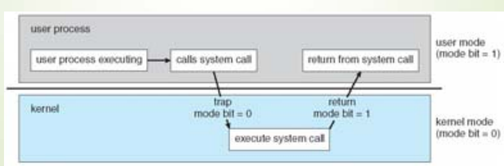
  Different machines/OS communicate closely enough to provide the illusion that there is only one system.

Operating-System Operations

- Dual-mode operation allows OS to protect itself and other system components
  - "User" mode: Normal programs executed and
  - "Kernel" mode (or "supervisor" or "protected")
  - *What is needed in the hardware to support "dual mode" operation?*
    - A bit of state (user/system mode bit)
      - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions are designated to be privileged
  - For a privileged instruction:
    - System call changes mode to kernel
    - OS performs the activity on user behalf
    - Once done, resets the mode back to that of the user and continues

Operating-System Operations

- User → kernel transition sets the system mode AND saves the user PC
- Kernel → User transition clears system mode AND restores appropriate user PC



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (e.g., C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs:
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and
  - Java API for the Java virtual machine (JVM)

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- System call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Most details of OS interface are hidden from programmer by the API and is Managed by run-time support library

OS Design & Implementation

- Design and Implementation of OS is not "solvable", but some approaches have proven successful
  - Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **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

OS Design & Implementation (Cont.)

- Important to separate

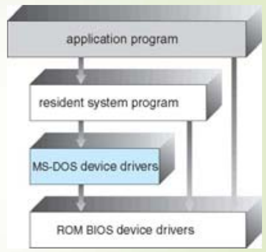
  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?
- Why to separate policy from mechanism?
  - allows maximum flexibility
  - policy decisions can be changed later (example – timer)
- Specifying and designing an OS is highly creative task of software engineering

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones:
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel -Mach

Simple Structure -- MS-DOS

- MS-DOS
- written to provide the most functionality in the least space
  - Not divided into modules
  - It may have some structure, but its interfaces and levels of functionality are not well separated



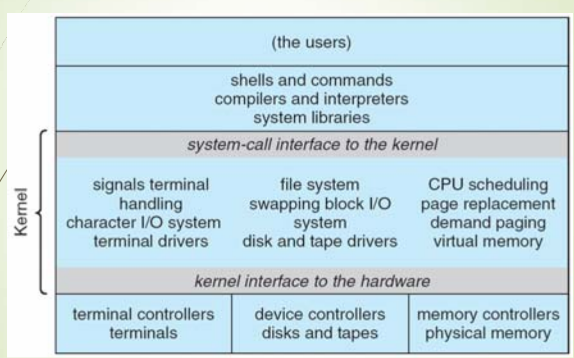
Non Simple Structure -- UNIX

limited by hardware functionality, the original UNIX operating system had limited structure. The UNIX OS consists of two separable parts

- Systems programs
- 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

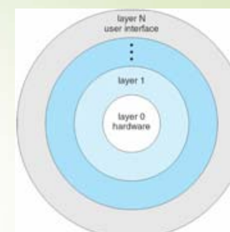
Traditional UNIX System Structure

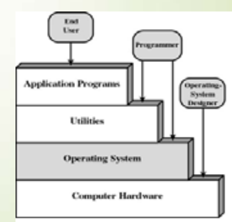
Beyond simple but not fully layered



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

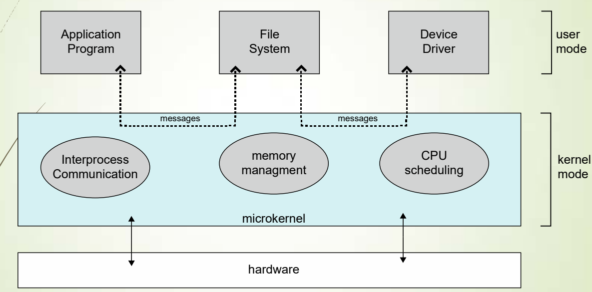




Microkernel System Structure

- Moves as much from the kernel into user space ➔ Microkernel
- Example - **Mach**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- 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

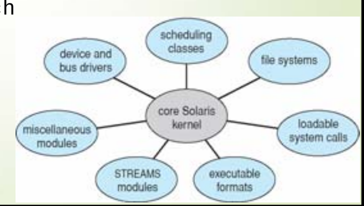
Microkernel System Structure



Modules

- Many 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.

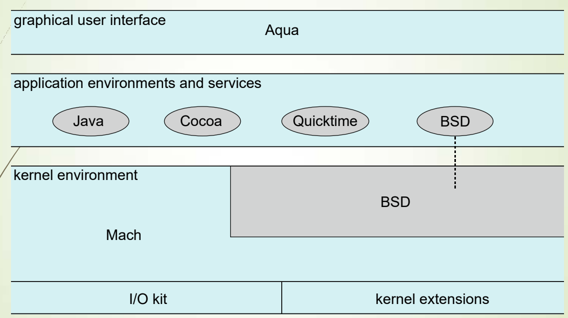
Solaris Modular Approach



Hybrid Systems

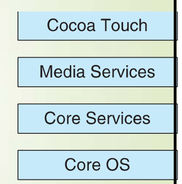
- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels use a combination of monolithic (efficient performance), modular (to add new functionality dynamically)
  - Windows is mostly monolithic (for performance), microkernel (to support different subsystems)
- Apple Mac OS X hybrid is layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure



iOS

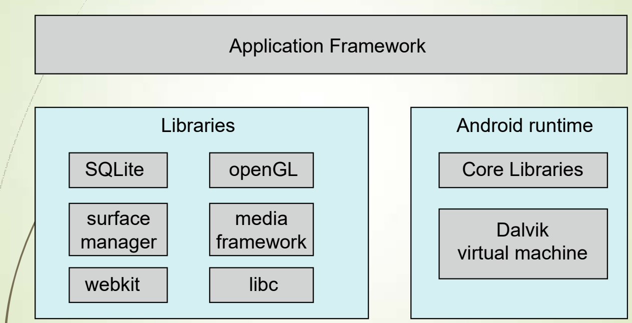
- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



Major Achievements in OS development

- Processes
- Memory Management
- Information protection and security
- Scheduling and resource management
- System structure

Processes

- Processes are the fundamental structure of operating systems
  - A process is a program in execution.
  - A unit of activity characterized by a sequential thread of execution, current state, and an associated set of system resources
  - Program is a *passive entity*, process is an *active entity*
- Process needs resources to accomplish its task
  - CPU, memory, I/O, 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

Processes (continued...)

- 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
- Processes solved the problems introduced by
  - Multiprogramming batch operations
  - Time sharing
  - Real-time transaction systems
- Principle tool available to system programmers in developing multi-tasking systems is the interrupt!

Processes (continued...)

- Processes consist of three components
  - An executable program
  - Associated data (variables, workspace, buffers, stacks, etc.)
  - The execution context of the program
- Coordination of processes is remarkably difficult
  - Improper synchronization
  - Failed mutual exclusion
  - Non-determinate program operation
  - Deadlocks

Processes Management Activities

- process management activities supported by the operating system:
  - Creation and deletion of both user and system processes
  - Suspension and resumption
  - mechanisms for process synchronization
  - mechanisms for process communication
  - mechanisms for deadlock handling

Memory Management

- Need:
  - All data to be in memory before & after processing
  - All instructions to be in memory to be able to execute
- Memory management involves
  - what is in memory, i.e., memory contents
  - Optimize CPU utilization
  - making computers to response to users

Memory Management (continued...)

- Principle storage management responsibilities
  - Process isolation
  - Automatic allocation/deallocation and management
  - Support of modular programming, i.e., deciding which processes (or parts thereof) and data to move into and out of memory
  - Protection and access control
  - Long-term storage
- These requirements typically met by
  - Virtual memory
  - File system facilities

Chapter 3
Processes

Process Concept

- Computers can do several activities at a time
  - Executing user programs,
  - reading from disks
  - writing to a printer, etc.
- In **multiprogramming:**

  Single CPU but switches from program to program - **pseudo parallelism or virtual CPU**
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**

  The terms ***job*** and ***process*** is used interchangeably.

Process Concept

- **File** – A passive entity -
- **Process** – a program in execution; process execution must progress in sequential fashion
  - An active entity
  - Needs resources such as CPU, memory, I/O
  - Resources are allocated:
    - at the beginning or
    - on demand during execution.
  - At any given point in time, system consists of:
    - System processes
    - User processes

  ➔ all exist **concurrently**.

Process Concept (Cont.)

- OS is responsible to:
  - Create process
  - Delete process
  - processes scheduling
  - Provide mechanisms for
    - Process synchronization,
    - Inter-process communication and
    - deadlock handling.
  - Traditionally
    - a process contains a single thread of controls
    - most modern OS support processes with multiple threads

Process Concept (Cont.)

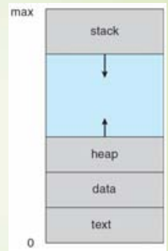
- A process includes:
  - A program counter (PC)
  - CPU registers:

    for data manipulation
  - Stack:

    for function parameters, return address, etc.
  - Data section:

    for global variables
  - Heap:

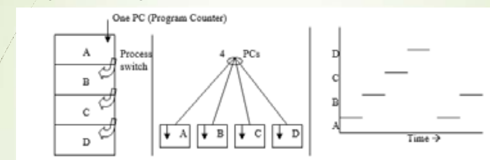
    for dynamic allocation of memory during run time.



- Can there be two processes associated with the same program?
  - Yes and will be considered two executions instances.

Process Concept (Cont.)

- **Abstract View:** each process has its own CPU, running independently



- Over period of time, all processes have made progress but in reality, at any given point in time, **only one** process is active.

Process Concept (Cont.)

Important: with switching back and forth,

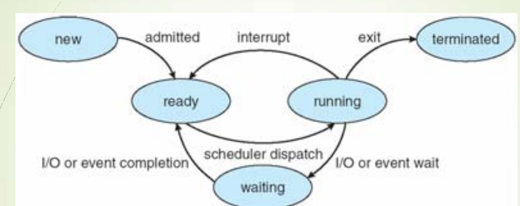
- Process computation rate may not be uniform.
- time may not be reproducible even with same set of processes.

→ **no assumption** should be made about built-in timings.

Generally, most processes are not affected by underlying multiprogramming of CPU or relative speed of other processes

Process State

- As a process executes, it changes **state**



- **new**: process is being created
- **running**: Instructions are being executed
- **waiting**: process is waiting for some event to occur (I/O completion or reception of a signal
- **ready**: process is waiting to be assigned to a processor
- **terminated**: process has finished execution

Process Control Block (PCB)

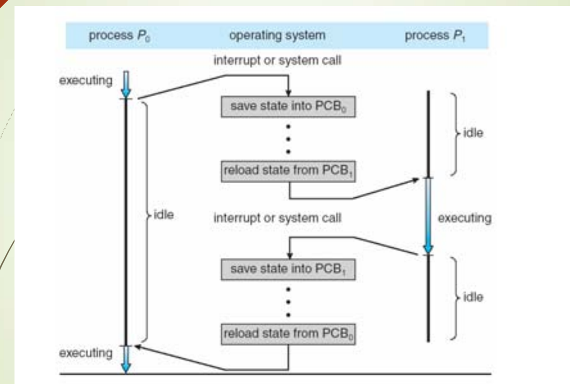
Information associated with each process

(also called **process/task control block or PCB**)

- 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
- Pointer - point to the PCB of another process in the list

Pointer	Process state
Process # (PID)	
Program Counter	
Registers	
Stack Pointer	
Memory limits	
List of open files	
Process start time	
CPU time	
:	
:	

CPU Switch From Process to Process



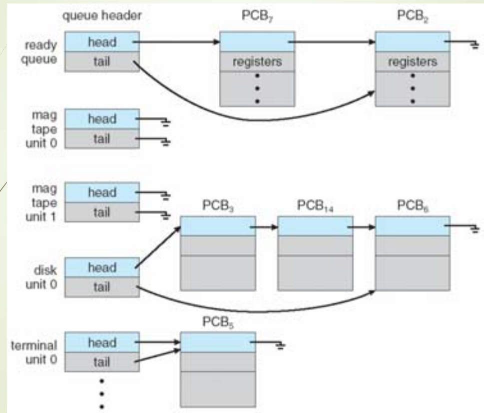
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Will talk more about it in next chapter

Process Scheduling

- **Uni-process system:**
  - No problem since there is only one process
- **Multi-process system:**
  - Process waits until CPU is free (either because of I/O or time quantum expiration)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
- During its life, a processes migrate among various queues

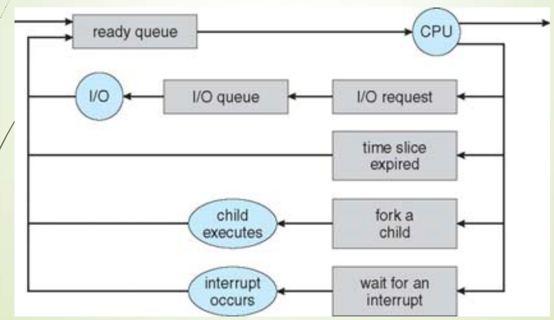
Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources and flows

- Rectangles: Queue
- Circles: Resources that serve queues

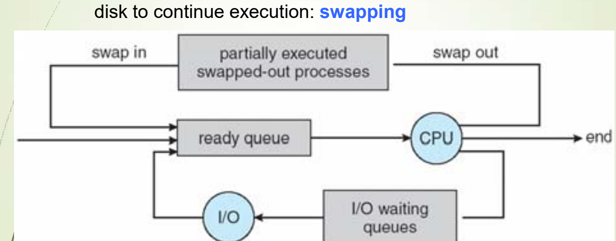


Schedulers

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

Addition of Medium Term Scheduling

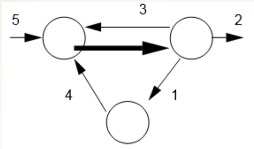
- if degree of multiple programming needs to decreased – **Medium-term scheduler**
  - Removes processes from memory, store on disk, bring back in from disk to continue execution: **swapping**



Advantage: may improve job mix, free up limited memory

Scheduling (contd.)

- When does a scheduler executes?



1. a process switches from running to waiting
   - ➔ select new process for CPU
2. a process terminates
   - ➔ select new process for CPU
3. a process switches from running to ready
   - ➔ select new process for CPU
4. a process switches from waiting to ready
5. a new process

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
  - **Foreground:**
    - A single process - controlled via user interface
  - **Background:**
    - multiple 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 keep running even if background process is suspended
  - Service has no user interface but 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 new process via a **context switch**
- **Context** of a process is represented in the PCB
- Context-switch time is a pure overhead;
  - CPU is busy switching between processes ➔ bottle neck?
  - Effects of more complex OS and the PCB?
- Time depends on hardware support

Operations on Processes

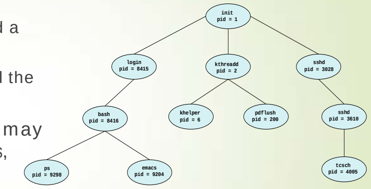
- process creation,
- process execution
- process termination,
- …

Operations on Processes

- process creation,
- process execution
- process termination,
- …

Process Creation

- During execution, a process may create several new processes, via a **create-process** system call.
  - The creating process is called a **Parent** process,
  - the new processes are called the **children** of that process.
- Each of these new processes may in turn create other processes, forming a **tree** of processes
- In most OS including UNIX and Windows family, Processes are identified by a unique id, called **Process ID (PID).**



Process Creation (Cont.)

- Every process needs resources. How to allocated resources?

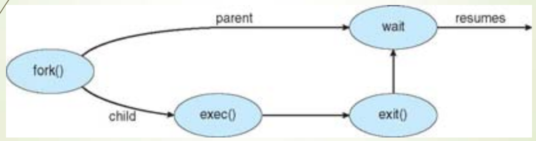
  3 possible scenarios

  1. Parent and children share all resources
  2. Children share only a subset of parent’s resources
  3. Parent and children share no resources. In this case, a child process may ask for its own resources from OS
     - It can get only a subset of the parent's resources.
     - **Why**?

       Prevents processes from overloading system

Process Creation (Cont.)

- UNIX examples
  - A new process is created using **fork()** system call
  - Parent and child share same address space
  - Both continue after **fork()** system call.
  - A return code: **fork()** == 0 for child and 1 for parent.
  - **execv()** system call is used by one of the two processes to replace the process's memory space with a new program.



Process Execution

Execution: 2 possibilities

1. Parent continues to execute concurrently with its children
2. Parent waits until some or all of its children are terminated

Process Termination

Process can termination either normally or abnormally

- How a process can terminate?

1. By itself
   - A process terminates itself.
   - Resources are de-allocated.
2. A parent can terminate a child process (via **abort()** system call)
3. **WHY**?
   - A child has exceeded its limits for usage resources
   - The task is no longer required
   - The parent is exiting
     - **Cascaded termination:** All children, grandchildren, etc. are terminated.
     - OS initiates the termination

Process Termination (contd.)

3. The parent process may wait through the **`wait()`** system call.

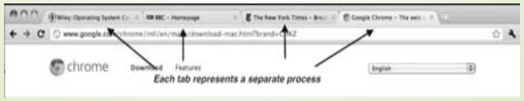
```
pid = wait(&status);
```

- If a process is terminating and its parent is not waiting for its termination, the process is called a **zombie** process
  - All processes transition to this state when they terminate but generally they exist as zombies **ONLY** briefly
- **orphan** process

- **UNIX**
  - Child terminates by **`exit()`** system call
  - **`wait()`** returns the process identifier of the child process so parent can identify its child
  - If parent terminates, children are assigned a **new parent**
    - the '**`init`**' process.

Multiprocess Architecture – Chrome Browser

- Many web browsers use to run as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Most modern browsers such as Google Chrome Browser is multi-process 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 is 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



Interprocess Communication

- Will talk about it when we start chapter on **Process Synchronization** (Chapter 5)

Interprocess Communication

- Two types of Processes:
  1. ***independent***
  2. ***cooperating***
- **Independent Processes:**
  - Cannot affect or be affected by the execution of any other process.
  - Do not share data with any other process
  - Deterministic execution
  - Reproducible execution
  - Execution can be stopped / started without any problem
- **Cooperative Processes:**
  - Can affect or be affected by the execution of any other process.
  - State /data is shared among other processes
  - Result of execution cannot be predicted in advance
  - Non-deterministic execution.

Interprocess Communication (contd.)

Why do we allow process co-operation?

- Information sharing
  - processes may be interested in same piece of information
  - Computation speed-up.
    - Given a task, break it into subtasks
      - Each subtask is running in parallel (possible if multiple CPUs or I/O channels)
- Modularity
  - develop a system in modular for allocation to separate processes or threads.
- Convenience
  - Individual users may be doing several activities, e.g., compiling a program, reading mail, etc.

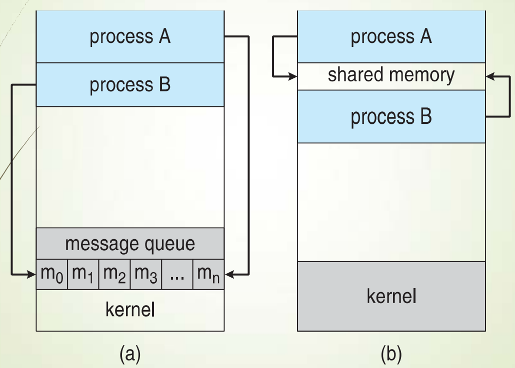
Interprocess Communication (contd.)

Examples:

- **Producer-Consumer** problem:
  - **Producer** produces the item
  - **consumer** consumes the item
    - compiler → assembly language code → object modules
    - web server may be a producer and client a consumer.
- To implement, need:
  - Mechanisms for process communication **interprocess communication** (**IPC**)
  - Methods for process synchronisation.
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

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



Communications Models

Differences and ad/disadvantages:

- In shared-memory model
  - a region of memory shared by cooperating processes is established. Processes can then exchange information this shared region.
  - Easy to program since on same hardware.
  - Difficult to scale up with number of processes
  - allows maximum **speed** and convenience of communication
- How to establish shared-memory systems?
  - Through **system calls**.
  - all accesses are same as routine memory accesses → no assistance from the kernel.

Communications Models (contd.)

message-passing model

- communication is through exchange of messages between processes.
- typically implemented **using system calls**
  - **slower** then Shared memory due to kernel intervention.
- useful for exchanging **smaller amounts of data**
  - no conflicts to be avoided.
- **easier to implement**

Communications Models (contd.)

How to establish Interprocess Communication in Shared-Memory Systems?

- Requires establishing a region of shared memory
  - Typically region resides in the address space of the process creating the shared-memory segment.
- processes wishing to communicate using this shared-memory segment need to attach it to their address space.
  - *Normally, OS system tries to* ***prevent*** *one process from accessing another process's memory but shared memory requires that two or more processes* ***agree to remove this restriction***. They can then exchange information by reading and writing data in the shared areas.
- The **form of the data** and the **location** are determined by these processes and are **not under the operating system's control**.
- The processes are responsible to ensure that they are not writing to the same location simultaneously.

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
- **Bounded-Buffer – Shared-Memory Solution**
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

36

Bounded-Buffer – Producer/Consumer Code

Producer code

```
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;
}
```

Consumer code

```
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 */
}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

37

Interprocess Communication – Message Passing

Message Passing Systems

- Distributed memory
- Multiple processors
- Common in distributed systems
- Need to establish a link and protocols
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

38

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:
  - What should be the size of message?
  - 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?

39

Message Passing (Cont.)

1. Size of message
   1. Variable length:
      - easy for the user / hard for system and to implement.
   2. Fixed Length:
      - harder for user / easy to implement.
2. How to establish communication link?
   1. unidirectional link or bi-directional link
   2. link capacity
   3. Link association between more than two processes.
3. How does a process know the id of the other?

Message Passing (Cont.)

4. Lost Messages:
   - Need an acknowledgement
   - Re-transmit if no acknowledgement after fixed time.
   - Receiver should have a method of identifying duplicates.
5. To block or not to block after sending a message!!! i.e., synchronous or asynchronous
   - Sender:
     1. Blocks on send() until an acknowledgement is received.
     2. Doesn't block.
   - Receiver: Blocks until a message is received (i.e. someone does a send())

Message Passing (Cont.)

6. Direct / Indirect Communication
   - Direct: Must explicitly name recipient
     - Send(P, msg): send message to process P
     - Receive(Q, msg): receive message from process Q
   - **Symmetric:** both sender and receiver have to name each other
     - send(P, message) —Send a message to process P.
     - receive(Q, message) —Receive a message from process Q.
   - **Asymmetric:** only sender names recipient and the receiver receives message from any sender.
     - send(P, message) —Send a message to process P.
     - receive(message) —Receive a message from any process.

Message Passing (Cont.)

- Indirect
  - Messages are send to and received from mailboxes (ports)
  - Each mailbox has unique id
  - Two processes can communicate only if there is a shared mailbox
    - send(A, message) —Send a message to mailbox A.
    - receive(A, message) —Receive a message from mailbox A.

Message Passing (Cont.)

- Mailbox sharing
  - *Let P_1, P_2,* and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who will get 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.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** OR **synchronous**
  - **Blocking send** – sender is blocked until message is received
  - **Blocking receive** -- receiver is blocked until a message is available
- **Non-blocking** OR **asynchronous**
  - **Non-blocking send** -- sender sends the message and continues
  - **Non-blocking receive** -- receiver receives:
    - A valid message, or
    - Null message
- Different combinations are possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

Producer-consumer becomes trivial

Produce code

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

- Produce code

```
message next_consumed;
while (true) {
   receive(next_consumed);
/* consume the item in next consumed */
}
```


Buffering

- How big is the queue of messages attached to a link.
- implemented in one of three ways
  1. Zero capacity
     - rendezvous
  2. Bounded capacity
  3. Unbounded capacity

Chapter 4: Threads

Motivation

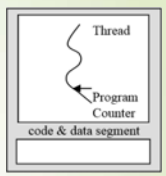
- a process is defined by two characteristics:
  1. An execution state and its location of execution
  2. Resources it uses
- A typical process has:
  - An address space
  - Protected access to:
    - CPU
    - other processes (for inter-process communication),
    - files and I/O resources.
- useful to share resources and accesse concurrently.
  - For example, a browser may be
    - display text/image,
    - receiving another request for a page,
    - retrieve data from network
    - another one to print a web page.

Motivation (contd.)

- **Traditional approach:**

  Single process - concept of thread is not recognized

  - process is called a **heavyweight process.**

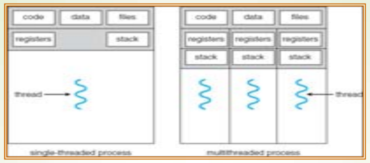


- In modern operating Systems, the Unit of CPU utilisation or dispatching is called **thread.**
  - consists PC, register set, and stack space.
- In a multiple process model, each process operates independent of others → useful for unrelated.
- Multiple processes performing the same task ➔ less efficient.

Motivation (contd.)

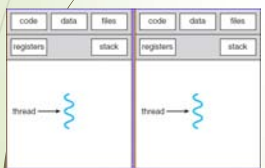
- **Multithreading** is the ability of an OS to support multiple threads of execution within a single process.
  - Shares with other peer threads its code and data sections, and other OS resources.
- Threads run within application and multiple tasks within the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation → heavy-weight operation
- Thread creation → light-weight operation
  - Can simplify code ➔ increases efficiency
- Kernels are generally multithreaded

Threading

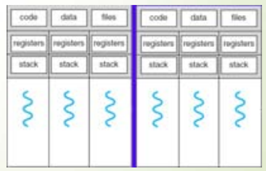


One process one thread
e.g. MS-DOS

One process. multiple threads
e.g. Java run-time engine



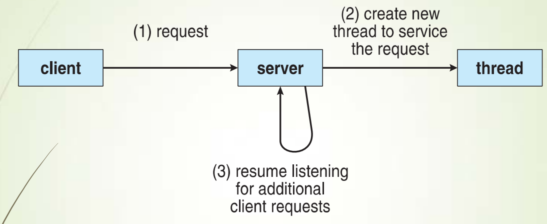
Multiple processes
one thread / process, e.g. UNIX



Multiple processes, multiple threads / process
e.g., Solaris, Linux, NT, Mach, Windows 10, IOS, etc.

5

Multithreaded Server Architecture



6

Benefits

- **Responsiveness –** execution may continue even if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process
- **Economy –** due to extensive resource sharing
  - Creation of a new thread in an existing process
    - Solaris: 30 times faster to create a new thread than creating a new process.
    - Context switching is 5-times faster.
  - Termination of a thread
  - Switching between threads within the same process
    - a thread context switch still requires a register set switch but no memory-management-related work need to be done.
  - communication efficiency between different executing programs.
    - No kernel intervention for protection / communication.
- **Scalability –** process can take advantage of multiprocessor architectures
  - Multithreading on a multi-CPU machine or multi-CORE processor increases parallelism

7

Multicore VS Multi-Processor Systems

- A multi-core architecture is more efficient than a multi-processor system because of:
  - faster on-chip communication
  - Uses less power.
  - Each core has its own set of registers as well as own cache (or can also use shared cache). These N-cores appear to the OS as N standard processors.
  - Economy

8

Thread Model

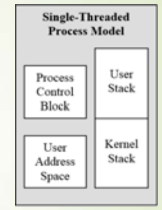
A process may have one or more threads with following:

- A thread execution state
  - Running, ready, blocked, terminated
- A saved thread context when not running;
  - Think of a thread is as an independent program counter operating within a process.
- An execution stack.
- Some per-thread static storage for local variables
- Access to memory and resources of its process, shared with all other threads.

9

Thread Model (contd.)

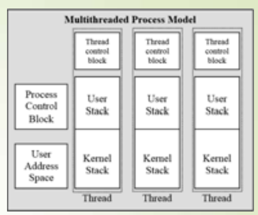
In a single threaded process model, the representation includes:



- Process control block (PCB)
- User address space
- User and kernel stacks to manage the call/return behaviour of the executing process.
- While process is running, processor registers are controlled by that process
- Contents of these registers are saved when the process is not running

10

Thread Model (contd.)



In a multithreaded environment:

- Single process control block
- User address space
- Separate stacks for each thread
- Separate control block for each thread containing register image, priority, and other thread related state information.

All threads of a process:

- Share the state and resources of the process
- Reside in the same address space and have access to same code/data.
- If a thread alters an item of data in memory, other threads see the results if and when they access that item.
- If a thread opens a file with read privileges, other threads in same process can also read from that file.

11

Multicore Programming

- challenges for programmers for a **Multicore** or **multiprocessor** systems
  - **Dividing activities**
  - **Balance**

    tasks perform equal work
  - **Data splitting**

    Just like applications, data accessed and manipulated by the tasks must be divided to run for such separate tasks
  - **Data dependency**

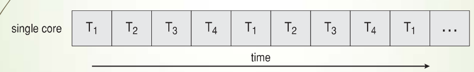
    Data accessed by tasks must be examined for dependencies between two or more tasks. For cases of dependencies, the tasks needs synchronization.
  - **Testing and debugging**

    Testing and debugging of these paths is inherently more difficult

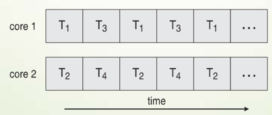
12

Multicore Programming

- ***Parallelism*** implies a system can perform more than one task simultaneously
- On a single core system, concurrency means inter-leaving of threads execution.
- On a multi-core system, concurrency → threads can run in parallel.
- **Concurrent execution on single-core system:**



Parallelism on a multi-core system



13

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Oracle SPARC T4 has 8 cores, and 8 hardware threads per core

14

Thread Functionality

A Thread has:

- Execution State
- May synchronise with one another

15

Thread Functionality (contd.)

Thread States:

- **Spawn:**
  - Typically when a new process is spawned, a thread for that process is also spawned.
  - A thread within a process may spawn another thread within the same process. The new thread gets an instruction pointer and arguments.
  - New thread gets its own register context and stack space and is placed on the ready list.
- **Block:**
  - When a thread needs to wait for an event, it will block (saving its user registers, PC and SP).
  - Processor may now turn to the execution of another ready thread.
- **Unblock:**
  - When the event for which a thread is blocked occurs, the thread is moved to the ready state.
- **Finish:**
  - When a thread completes, its register context and stacks are de-allocated.

16

Thread Functionality (contd.)

Thread synchronisation:

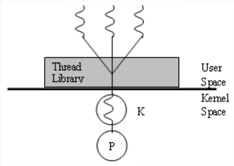
- All threads of a process share the same address space and other resources, such as open files.
  - Alteration of a resource by one thread affects the environment of the other threads in the same process.
- Necessary to synchronise the activities of various threads so that they do not interfere with each other or corrupt data structures.
  - **For example,** two threads try to add an element to a double linked list, one element may be lost or the list may end up malformed.
- **Many different methods** and techniques - same as for processes. We will talk about these later when talking about process synchronization (next chapter).

17

User-level Threads & Kernel-level Threads

Two broad categories of implementation:

- **User-level threads (ULT)**
  - Thread management is done by the application



  - An application can be programmed to be multithreaded by using a thread library.
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads

User-level Threads

- What a Thread library provides?
  - Code for creating and destroying threads
  - Code for message passing between threads
  - Code for scheduling thread execution
  - Code for saving and restoring thread contexts.

User-level Threads (contd.)

An application, by default, starts as a single thread

- Kernel allocated resources to a single process and singl thread.
- In running state, the process can spawns a new thread
  - runs within the same process
- Spawning is done using the spawn utility.
- Control is passed to that utility by a function call.
- Thread library is responsible to:
  - creates a data structure for the new thread
  - pass control to one of the threads within the process that is in ready state using some scheduling algorithm.

User-level Threads (contd.)

- When control is passed to the library, the context of the current is saved, and when control is passed from library to the thread, the context of that thread is restored
- Context consists of:
  - Contents of user registers
  - Program counter
  - Stack Pointer
- All of this activity takes place in user space and within a single process.
- Kernel continues to schedule the process as a unit and assigns a single execution state.

User-level Threads (contd.)

Advantages:

- Thread switching doesn't require kernel mode privilege.
  - saves the overhead of two mode switches → switching is fast.
- Scheduling can be application specific.
- One application might benefit most from simple Round-Robin scheduling algorithm while other might benefit from priority-based scheduling.
- Scheduling algorithm can be modified **without** disturbing underlying OS scheduler.
- User-level threads (ULT) can run on any operating system. No changes are required to change the underlying kernel to support ULT.

22

User-level Threads (contd.)

Disadvantages:

- typically, most system calls are blocking so what is the effect of system call?
  - When a thread executes a system call, not only that thread but the entire process is blocked.
- In a pure User-level Thread system, a multithreaded application cannot take advantage of several cores/CPU system.
  - A kernel assigns one process to only one processor at a time, i.e., only a single thread within a process can execute at a time.

23

Kernel-level Threads

Kernel-level threads (KLT)

- All thread management is done by the kernel.
- application-programming interface (API) to the kernel thread facility
- Examples – virtually all general purpose operating systems:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

24

Kernel-level Threads (contd.)

- An application can be programmed to be multithreaded.
- All threads within an application are supported within a process.
- Kernel maintains the context information for the process as a whole and individual threads.
- Scheduling by the kernel is done on a thread basis.
- Overcomes two main drawbacks for the User-Level Threads:
  1. Kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  2. If one thread in a process is blocked, the kernel can schedule another thread of the same process.

25

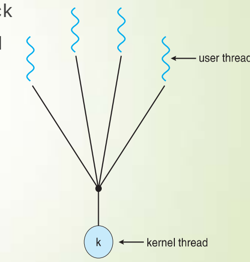
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

26

Many-to-One

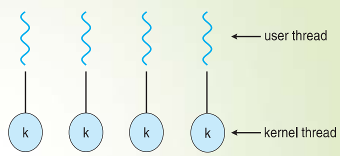
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



27

One-to-One

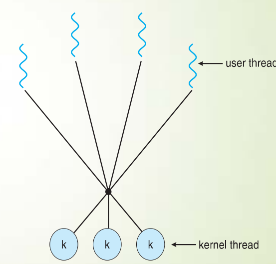
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



28

Many-to-Many Model

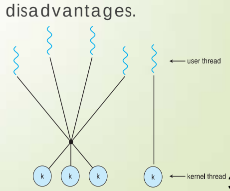
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



29

Two-level Model

- A variation of many-to-many model
  - multiplex many user-level threads to a smaller or equal number of kernel threads but also allow a user-level thread to be bound to a kernel thread, for example: Solaris
- Thread creation, scheduling and synchronisation of threads within an application are done in the user space.
- programmer can adjust number of Kernel-Level Threads for a particular application and machine for best results.
- Multiple threads within the same application can run in parallel on multiple processors.
- A proper design can combine the advantages of pure User or Kernel-Level Threads while minimising the disadvantages.
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



30

Threads

Important Questions:

1. *Does the blocking of thread results in the blocking of the entire process?*
2. *Does this prevent execution of any other thread in the same process even if that other thread is in a ready state?*

- Answer depends on whether it is a User-Level Threading or the Kernel-Level Threading. More effect in User-level threading.
- **Example:** Assume 2 processes, P0 and P1 such that process P1 is executing in its thread 3. Possible scenarios:
  1. Application executing in Thread 3 makes a system call (e.g., I/O call is made) ➔ P1 blocks
     - Control is transferred to the kernel.
     - Kernel invokes the I/O action and places process P1 in **wait state**.
     - Transfers control to process P0
     - According to the data structure maintained by the thread library, Thread 3 of process P1 is still in **running** state.
     - This process is not actually in running state but is perceived in the running state.

31

Threads

2. A clock interrupt passes control to the kernel.
   - Kernel determines that process P1 has exhausted its time quantum
   - places process P1 in the **ready** state and switches to process P0
   - According to the data structure maintained by the thread library, thread 3 of process P1 is still in **running** state.

32

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

33

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) defines development of API for thread creation and synchronization
  - It provides ***Specification*** but not ***implementation***
  - API specifies behavior of the thread library, implementation is up to development of the library
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example (Cont.)

```c
#include <stdio.h>
#include <pthread.h>

int gvar = 0;                    // define a global variable

void *myThreadFun(void *vargp){ // function to be executed by all threads

    int *myid = (int *)vargp;   // Store value of arg passed to this thread

    int lvar = 0;               // create a local var to observe its changes
    ++gvar;                     // update global variable
    ++lvar;                     // update local variable
                                // Print the argument, local and, global var
    printf("Thread ID: %d, local: %d, global: %d\n", *myid, ++lvar, gvar);
}

int main() {
    int i;
    pthread_t tid;

    for (i = 0; i < 3; i++)     // Let us create three threads
       pthread_create(&tid, NULL, myThreadFun, (void *)&tid);

    for (i = 0; i < 3; i++)     // wait for three threads to join
       pthread_join(tid, NULL);

    pthread_exit(NULL);
    return (0);
}
```


Pthreads Example (Cont.)

Explanation:
In main(): declare a variable called **thread_id** of type **pthread_t**, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.
pthread_create() takes 4 arguments.

- 1st argument is a pointer to thread_id which is set by this function.
- 2nd arg: specifies attributes. If NULL, default attr are used.
- 3rd arg: name of function to be executed for the thread to be created.
- 4th arg:used to pass arguments to the function myThreadFun.

pthread_join() function for threads is the equivalent of wait() for processes. A call to it blocks the calling thread until the thread with identifier equal to the first argument terminates.

How to compile above program?
To compile a multithreaded program using gcc/cc, need to link it with the pthreads library.

```
$ gcc thread_ex1.c -lpthread
$ ./a
Thread ID: 1184, local: 2, global: 1
Thread ID: 231184, local: 2, global: 2
Thread ID: 231184, local: 2, global: 3
$
```


Pthreads Example (Cont.)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum;
void *runner (void *param);

int main(int argc, char* argv[]){
  pthread_t tid;                     // thread identifier
  pthread_attr_t attr;               // set of thread attributes

  if (argc != 2){
    fprintf(stderr, "Usage: a.out <integer value>\n");
    return (-1);
  }

  if ((atoi(argv[1]) < 0)){
    fprintf (stderr, "%d must be >= 0\n", atoi(argv[1]));
  }

  pthread_attr_init(&attr);          // get default attributes

  // create a thread and wait for thread to exit pthread_join
  pthread_create(&tid, &attr, runner, argv[1]);
  printf("Thread ID: %d\n", tid);
  pthread_join(tid, NULL);

  printf("sum = %d\n", sum);
}
```

Pthreads Example (Cont.)

```
// thread will begin control in this function
void *runner(void *param){
  int i, upper = atoi(param);
  sum = 0;

  for(i = 1; i <= upper; i++)
    sum = sum + i;

  pthread_exit(NULL);

}
```

38

Pthreads Example (Cont.)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp){
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main(){
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);         //wait for thread to join
    printf("After Thread\n");
    exit(0);
}
```

OUTPUT Produced: **Before Thread**
Printing GeeksQuiz from Thread
After Thread

Source: https://www.geeksforgeeks.org/multithreading-c-2/

39

Thread examples

- Read at your own
  - 4.4.1: pthreads
  - 4.4.2: Windows Threads
  - 4.4.3: Java Threads

40

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
  DWORD Upper = *(DWORD*)Param;
  for (DWORD i = 0; i <= Upper; i++)
    Sum += i;
  return 0;
}

int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  if (argc != 2) {
    fprintf(stderr,"An integer parameter is required\n");
    return -1;
  }
  Param = atoi(argv[1]);
  if (Param < 0) {
    fprintf(stderr,"An integer >= 0 is required\n");
    return -1;
  }
```

41

Windows Multithreaded C Program (Cont.)

```
/* create the thread */
ThreadHandle = CreateThread(
  NULL, /* default security attributes */
  0, /* default stack size */
  Summation, /* thread function */
  &Param, /* parameter to thread function */
  0, /* default creation flags */
  &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
}
```

42

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface

43

Java Multithreaded Program

```
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

  public void setSum(int sum) {
   this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

44

Java Multithreaded Program (Cont.)

```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sumObject));
      thrd.start();
      try {
        thrd.join();
        System.out.println
               ("The sum of "+upper+" is "+sumObject.getSum());
     } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>"); }
}
```

45

Threading Issues:

- **The fork() and exec() System Calls**
  - If a thread in a program calls fork(), system call, does the new process duplicate all threads or is the new process single-threaded?
  - Some UNIX systems have chosen to have two version of fork() call, one that duplicates all threads and another that duplicates only the thread that invoked the fork() call.
  - If a thread invokes a exec() system call, what should be the behaviour?

46

Threading Issues:

Thread Cancellation:

- Thread Cancellation ➔ terminating a thread before it has completed its task.
  - For example, if multiple threads are concurrently searching a database and one thread return results, remaining threads might be cancelled. A thread that is to be cancelled is often referred to as the **target thread**.
- Cancellation may occur in two ways:
- **Asynchronous cancellation:** One thread immediately terminates the target thread.
- **Deferred cancellation:** target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly manner.
- Think of difficulty in situations
  - when resources have been allocated to a cancelled thread or
  - A thread is cancelled while in the midst of updating data it is sharing with other threads.

47

Threading Issues:

Signal handling:

- In UNIX, a **signal** notifies a process of occurrence of an event.
- A signal may be received:
  - **synchronously** (*signals delivered to the same process* that performed the operation that caused the signal)
  - **asynchronously** (*typically, sent to another process*) – depends on the source of and the reason for the event.
- All signals, whether synchronous or asynchronous, follow the same pattern:
  1. A signal is generated by the occurrence of a particular event.
  2. A generated signal is delivered to a process.
  3. Once delivered, the signal must be handled.
- **Examples of synchronous signals**: illegal memory access and division by 0.

48

Threading Issues:

Signal handling:

- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
  - Examples: terminating a process with specific keystrokes (such as <control><C>), timer expired.
- Every signal may be *handled* by one of two possible handlers:
  - A default signal handler
  - A user-defined signal handler
- Every signal has a **default signal handler** that is run by the kernel when handling that signal.
- Default action can be overridden by a **user-defined signal handler**.
- Signals may be handled in different ways. Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program

49

Threading Issues:

- Handling signals in single-threaded programs is straightforward; signals are always delivered to a process.
- more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?
- In general, the following options exist:
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process
- Method for delivering a signal depends on the type of signal generated

50

Implicit Threading

- More threads → more difficulty to ensure program correctness using explicit threads
- Typically, creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package

51

Thread Pools

- *Allowing a multithreaded server such as a Web server to create uncontrolled # of threads → **may** lead to problems:*
  - The amount of time.
  - No bound on the number of threads
- One solution to this issue is to use a **thread pool**.
  - create a number of threads at the start of the process and place them into a **pool** → sit and wait for work.
  - If request, awaken a thread from this pool
  - When work is done, return thread back to the pool
  - If the pool contains no available thread, the server waits until one becomes free.

52

Thread Pools

- **Advantages**:
  - faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- What should be the size of pool?
  - Decided heuristically, based on:
    - Number of CPU in the system
    - Size of memory
    - Expected number of concurrent requests
  - Sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns
    - provide benefit of having smaller pool

53

Chapter 5:

Process Synchronization

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classical Synchronization Problems
- Monitors
- Synchronization Examples
- Alternative Approaches (may be)

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completed execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Problem Illustration:
  We want a solution of the consumer-producer problem (introduced earlier) that fills ***all*** the buffers.
  - Introduce an integer `counter`
    - Will keeps track of the number of full buffers.
    - Initially, `counter = 0` → buffer is empty
    - Incremented by the producer after it produces a new item in buffer
    - Decremented by the consumer after it consumes an item from buffer.

Background

- **Producer-Consumer** problem (introduced earlier):
  - **Producer** produces the item
  - **consumer** consumes the item

Producer code

```
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;
}
```

Consumer code

```
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 */
}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Background

- We want a solution of the consumer-producer problem that fills ***all*** the buffers.
  - Introduce an integer **counter**
    - Will keeps track of the number of full buffers.
    - Initially, **counter = 0**
    - Incremented by the producer after it produces a new item in buffer
    - Decremented by the consumer after it consumes an item from buffer.

Producer – Consumer Problem

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer:

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;

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

Race Condition

- **counter++** in assembly language could be like:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Things which can create problems:
  - Cannot guarantee the **order** in which these assembly instructions will execute
  - Each process executes sequentially but we don't know **how they are interleaved**.
  - Each process continues running but we don't know at **what speed each runs**.

Race Condition

Assume counter = 5;

- Producer executes: counter++; | concurrently
- Consumer executes: counter--;
- What will be the value of counter?
  - 4, 5, or 6?
- Actual result: 5 is guaranteed only if both execute separately.

Counter = 5;
P: register1 ← counter;
register1 ← register1 + 1;
counter ← register1;
 	
counter = 6;
C: register2 ← counter;
register2 ← register2 -1;
counter ← register2;
counter = 5;
	No problem!

Race Condition

```
if:     Counter = 5;
        P:    register1 ← counter;
        C:    register2 ← counter;
        P:    register1 ← register1 + 1;          Problem!!!
        C:    Register2 ← register2 - 1;
        C:    counter   ← register2;
        P:    counter   ← register1;              counter = 6;
```

```
if:     Counter = 5;
        P:    register1 ← counter;
        C:    register2 ← counter;
        P:    register1 ← register1 + 1;          Problem!!!
        P:    counter   ← register1;
        C:    Register2 ← register2 - 1;
        C:    counter   ← register2;              counter = 4;
```

Race Condition

- Situations when two or more processes are writing to some shared data & the final result depends on **who runs precisely when** are called **race condition.**
- **Important**:
  - debugging programs with race condition is neither easy nor fun
  - Most of results are generally fine
  - ONLY once in a while something weird happens!

Solution: Need **mutual exclusion**

- Part of the time, a process may be busy doing internal computation and other things that do not lead to race condition and some other times, it may be accessing shared memory or files
- Part of the program where the shared memory is accessed is called **critical section.**
  - To avoid race condition, need to arrange that no two processes are in their critical section at the same time.
  - i.e., access to the shared resources should be **mutually exclusive**

Race Condition

- Situations when two or more processes are writing to some shared data & the final result depends on **who runs precisely when** are called **race condition.**
- **Important**:
  - debugging programs with race condition is neither easy nor fun
  - Most of results are generally fine
  - ONLY once in a while something weird happens!

Solution: Need **mutual exclusion**

- Part of the time, a process may be busy doing internal computation and other things that do not lead to race condition and some other times, it may be accessing shared memory or files
- Part of the program where the shared memory is accessed is called **critical section.**
  - To avoid race condition, need to arrange that no two processes are in their critical section at the same time.
  - i.e., access to the shared resources should be **mutually exclusive**

11

Critical Section

- General structure of a process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

12

Solution to Critical-Section Problem

1. **Mutual Exclusion**
   If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**
   If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** -
   A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at nonzero speed
   - No assumption should be made concerning **relative speed** of the ***n*** processes

13

Solution to Critical-Section Problem

4. **Similarity**
   All processes must be running identical critical section decision algorithm
5. **CPU Speed**
   No assumption should be made about either the relative speed or the number of processors in the system.

14

Two-process Solution

Assume two processes: P_0 & P_1

Solution 1:

```
Common variables: int turn;      // can be either 0 or 1

If (turn == i)  ➔ Process Pi can enter its CS
                  Pj denotes other process such that j=1 – i;
```

P_i:

```
Repeat
     Pre CS
     while (turn != i);
       wait;
     critical section
     turn = j;
     Post CS
```

Algorithm has:

- Mutual exclusion
- Progress ➔ NO
  - Processes are forced to alternate.
  - **Suppose** turn = 1 and process P1 is blocked for I/O, etc. What will happen?

Two-process Solution

Solution 2:

```
Common variables:    int turn;          // can be either 0 or 1
                     Boolean flag[2];
```

- initially `flag[0] = flag[1] = False`;
- if (`flag[i] == True`), then process **Pi** is ready to enter its CS

P_i:

```
Repeat
  Pre CS
  flag[i] = True;          // declares its intent
  while (flag[j] == True)   // checks if other is already in CS
       wait;
  critical section
  flag[i] = False;         // indicates its completion of CS
  Post CS
```

Algorithm has:

- Mutual exclusion
- No Bounded wait - May get involved into deadlock waiting to get into CS. How ?

```
P0: flag[0] = True;
P1: flag[1] = True;
P0: while (flag[1] == True) wait;
P1: while (flag[0] == True) wait;
```


Peterson’s Solution

Solution 3:

- It is a combination of solution 1 and 2.
- Common variables:

```
Common variables:  int turn;          // can be either 0 or 1
                   Boolean flag[2];
 Let, initially:
 flag[0]    = False;
 flag[1]    = False;
 if flag[i] == True,  then process Pi is ready to enter its CS
 if turn    == i,     then process Pi can enter its CS
```

P_i:

```
Repeat
     Pre CS
     flag[i] = True;  // Pi declares its intent
     turn = j;       // i gives j an opportunity to enter its CS
     while (flag[j] == True AND turn == j)
        wait;        // if j wants to enter & its j's turn, i must wait
     Critical Section;
     flag[i] = False;
     post CS
```


Peterson’s Solution

Observations:

- Pi first sets **`flag[i]`** to be true and then asserts it is other process's turn.
- If both processes try to enter their CS at the same time, turn will be either **i** or **j** roughly at the same time, i.e., only one of the two assignments will last (since the other one will be over-written).
- Essentially **`turn`** will decide which process enters CS first.

Proof of correctness:

- Observe that:
  - If process **Pi** is not in its CS, `flag[i] is` **`false`**.
  - If process **Pi** is in its CS, `flag[i] is` **`true`**.
  - **Case 1**: One of them, say **P1**, is away from CS (i.e., `flag[1] = False`)
    P0: tries to enter its CS and will succeed.
  - **Case 2**: One of them, **P1** is in CS
    ➔ **P0** reaches the while loop, thus
    `turn == 1; and flag[1] == True;`
    ➔ **P0** waits.
    Therefore, mutual exclusion is guaranteed.

Peterson's Solution

- **Case 3**: both P_0 and P_1 are trying to enter
- Let **S(i)** → statement process #i is executing, then we have the following scenario:
- **Case i:** `S(0) > S(1)` or vice versa
- **Case ii:** `S(0) == S(1)` (i.e., same statement)

  Assume both are executing **statement 3**

  Let: `turn == 1;`

  → Process P_0 goes first to execute the while loop

  Does a busy wait → blocks due to time out.

  → Process P_1 will succeed in entering CS

Peterson's Solution

- **Progress: Pi** can be prevented from entering CS only if it is stuck in its while loop with:

  `flag[j] == True AND turn == j;`

  1. If P_j is not ready to enter its CS, flag[j] == False → **Pi** can enter its CS
  2. If P_j has set (flag[j] == True) and is in while loop, then either

     `turn == i;` or `turn == j;`

     Pi will enter CS **Pj** will enter CS

  - Once **Pj** exits CS, it will reset `flag[j]` to `False` → **Pi** will enter its CS.
- **Bounded Wait:** After at most one entry by **Pj**, **Pi** will also enter its CS.

Multi-process Solution (not in book)

- **Multiple Process Solution:** Involves tokens. Assume n processes

```
Common variables: {idle, want_in, in_CS} states of execution
                  flag[n] = idle;
                  int turn;
Pi: int j;
    Repeat
         flag[i] = want_in;
         j = turn;
         while(j != i)              // move token clockwise
          if (flag[j] != idle)      // ensure no one is ahead.
            j = turn;
          else
            j = (j + 1) mod n;
         flag[i] = in_CS;
         j = 0;
         while( (j<n) && (j == i || flag[i] != in_CS) )
           j++;
    Until ((j >= n) && (turn == i || flag[turn] == idle) );
    turn = i;
    Critical Section
    j = (turn + 1) mod n;     // any process willing to enter CS will
    while ((flag[j] == idle)  // after at most n-1 turns.
      j = (j + 1) mod n
    turn = j;
    flag[i] = idle;
    post CS
```

- if several processes want_in, closest one will get_in.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors → disable interrupts
  - Currently running code would execute without preemption
- Disabling interrupt not feasible in multiprocessor environment
  - Too much message passing overhead for disabling interrupt
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

23

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ;    /* do nothing */

    Critical Section

    lock = false;
            /* remainder section */
 } while (true);
```

- If two instruction are executed simultaneously, they will be executed sequentially in some arbitrary order.
- Provides Mutual exclusion
- Does **not** satisfy bounded wait.

24

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
  return temp;
 }
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   Critical Section
   lock = 0;
    /* remainder section */
} while (true);
```

- First process that invokes **compare_and_swap** will set lock to 1 and enter CS since original value of **lock == expected**, i.e., 0
- Subsequent call to **compare_and_swap** will NOT succeed since **lock != expected**.
- When a process exits its CS, it sets lock back to 0.
- Satisfies Mutual exclusion
- Does **not** satisfy bounded wait.

Bounded-waiting Mutual Exclusion with test_and_set

Do it by yourself

- Common data structures: Boolean waiting[n]; // initialized to false
  Boolean lock; // initialized to false

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   Critical Section
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

- Satisfies all of the Critical Section requirements.

Mutex Locks

- Previous solutions are generally complicated for application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- This solution requires **busy waiting**
  - therefore called a **spinlock**

acquire() and release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}
do {
acquire lock
   critical section
release lock
  remainder section
} while (true);
```

29

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore *S* – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

30

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore "**synch**" initialized to 0

P1:

   S_1;

   signal(synch);

P2:

   wait(synch);

   S_2;

- Can implement a counting semaphore *S* as a binary semaphore

31

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- The semaphore require busy-waiting
  - Wasting CPU time
  - Not a good idea, especially for n-processes or when may spend lots of time in critical section
- Also called **spin lock**.
- Useful on multiprocessor systems since there is no context switch.
- Can modify the definition of wait and signal such that:
  - When a process executes wait and finds a semaphore **S** value **not positive**, it can block itself by a wait call on semaphore S

32

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – places a process into a waiting queue associated with semaphore, switching the state of process to be waiting ➔ CPU picks and executes another process
    - A waiting process blocked on S can be restarted by some other process-executing signal
  - **wakeup** – removes one of processes in the waiting queue and place it in the ready queue
    - Process is restarted by **wakeup** operation and its state is changed from wait to ready

33

Implementation with no Busy waiting (Cont.)

```
typedef struct{
     int value;
     struct process *list;
   } semaphore;

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();                    // block this process
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {           // if no one is waiting, no problem
      remove a process P from S->list;
      wakeup(P);                  // wakeup this process
   }
}
```

34

Deadlock and Starvation

- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	...
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended.
  - If processes are added and removed from waiting queue in LIFO format.

35

Implementing Binary semaphores

- A semaphore with an integer value ranging between 0 and 1;
- Counting semaphores can be implemented using binary
- Let we have following variables:

```
binary_semaphores       S1, S2
int                     C;
Let:                    init(S1, 1);
                        init(S2, 0);

wait(S): wait on counting semaphore S
          wait(S1);
            C--;
         if (C < 0)
          signal(S1);
           wait(S2);
         signal(S1);

signal(S): signal on counting semaphore S
          wait(S1);
            C++;
         if (C <= 0)
          signal(S2);
            else
          signal(S1);
```


Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
    - First Readers-Writers Problem
    - Second Readers-Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

Same as before

- Producer producing an item that it placed in buffer. If buffer is full, it should wait.
- Consumer consumes an item already in buffer. If nothing in buffer, it should wait.

- Number of buffers: ***n***, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
       ...
        /* produce an item in next_produced */
       ...
     wait(empty);
     wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
     signal(mutex);
     signal(full);
  } while (true);
```

Bounded Buffer Problem (Cont.)

- structure of the consumer process

```
Do {
     wait(full);
     wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
     signal(mutex);
     signal(empty);
        ...
     /* consume the item in next consumed */
        ...
 } while (true);
```

40

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write

1. Only readers
   - no problem. Example: all users trying to execute word processor or compiler
2. Only writers
   - Allow only one at a time, old Critical Section/Mutual Exclusion problem. Many clients trying to update the file at the same time
3. Both readers and writers:
   - Problem. For example, travelling agencies accessing for seat reservation.
   - Writers should have exclusive access to shared object and Only one single writer can access the shared data at the same time

- We are interested in case 3 of the above.
- Different variations:

41

Readers-Writers Problem

First reader-writers.

- No reader be kept waiting unless a writer has already obtained permissions to use the object.
  - i.e., no reader should wait for others readers to finish simply because a writer is waiting.

Second readers-writers.

- Once a writer is ready, the writer performs its write a.s.a.p.
  - i.e., if a writer is waiting to access object, no new reader may start reading.
- Either of the two solutions lead to **starvation**.
- Solution to the first readers-writers problem

Solution 1:

Initialization:

```
init(wrt, 1);
```

Writer:	**Reader:**
`wait(wrt);`	`wait(wrt);`
`do the writing`	`do the reading;`
`signal(wrt);`	`signal(wrt);`

- **Problem:** allows only one reader at a time ➔ bad solution

42

Readers-Writers Problem

Solution 2:

- **Common** semaphore variables: `mutex, wrt;`
  Integer variables: `read_count = 0;`
- **Initialization:** `init(mutex, 1); init(wrt, 1);`
- **Writer:**

```
wait(wrt);
Do the writing;
signal(wrt);
```

- **Reader:**

```
wait(mutex);
read_count++;
if(read_count == 1)
  wait(wrt);   // if writer is already there, wait. If not,
signal(mutex); // writer will wait until all readers are done
    :
    read;
    :
wait(mutex);
read_count--;
if(read_count == 0)
  signal(wrt);
signal(mutex);
```

43

Dining-Philosophers Problem

- A classical synchronization problem.
- Represents a large class of concurrency control problems.

- 5 philosophers at a round table.
- 5 chopsticks on the table - one between each of the philosophers.
- A bowl of rice at the center of the table.
- Philosophers alternate between eating and philosophising.
- In order to eat, a philosopher needs both left and right chopstick.
- Can pickup only closest chopstick and one at a time (and only from the table – need to behave).
- When a hungry philosopher has both chopsticks – (s)he can eat without releasing chopstick.

44

Dining-Philosophers Problem

- When finished eating, puts down both the chopsticks & starts thinking.
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)

    Common variables: `semaphore chopstick[5];`
    Initialize : `all initialized to 1;`

```
Philosopher i:
  Repeat
     wait(chopstick[i]);
     wait(chopstick[i+1]%5);
     eat();
     signal(chopstick[i]);
     signal(chopstick[i+1]%5);
     think();
```

45

Dining-Philosophers Problem Algorithm

- Solution guarantees that no two neighbours are eating simultaneously.
- What is the problem with this algorithm?
  - May lead to deadlock.

46

Dining-Philosophers Problem Algorithm (Cont.)

- Need to impose ordering to avoid deadlock. Possible solutions include:
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available
    - right to eat move in a circle, i.e., token passing.
  - Use an asymmetric solution
    - an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- **IMPORTANT**: A deadlock free solution doesn't always guarantees elimination of starvation.

47

Dining-Philosophers Problem

- A classical synchronization problem.
- Represents a large class of concurrency control problems.

- 5 philosophers at a round table.
- 5 chopsticks on the table - one between each of the philosophers.
- A bowl of rice at the center of the table.
- Philosophers alternate between eating and philosophising.
- In order to eat, a philosopher needs both left and right chopstick.
- Can pickup only closest chopstick and one at a time (and only from the table – need to behave).
- When a hungry philosopher has both chopsticks – (s)he can eat without releasing chopstick.

Dining-Philosophers Problem

- When finished eating, puts down both the chopsticks & starts thinking.
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)

    Common variables: `semaphore chopstick[5];`

    Initialize : `all initialized to 1;`

```
Philosopher i:
  Repeat
     wait(chopstick[i]);
     wait(chopstick[i+1]%5);
     eat();
     signal(chopstick[i]);
     signal(chopstick[i+1]%5);
     think();
```


Dining-Philosophers Problem (contd.)

- Solution guarantees that no two neighbours are eating simultaneously.
- What is the problem with this algorithm?
  - May lead to deadlock.

Dining-Philosophers Problem (Cont.)

- Need to impose ordering to avoid deadlock. Possible solutions include:
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available
    - right to eat move in a circle, i.e., token passing.
  - Use an asymmetric solution
    - an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- **IMPORTANT**: A deadlock free solution doesn't always guarantees elimination of starvation.

Problems with Semaphores

- Convenient and effective mechanism but tricky to use.
- Burden of synchronisation on programmer. Incorrect use of semaphore operations:

```
1. signal(mutex)        2. wait(mutex)
        :                     :
   CS                    CS
        :                     :
   wait(mutex)           wait(mutex)
```

No mutual exclusion — Deadlock

- Suppose a process omits the wait(mutex)or the signal(mutex)or both. In this case, either there is no mutual exclusion or deadlock will occur.
- Errors could be due to:
  - Honest programming errors
  - Non-cooperative programmer

Monitors

- Monitors are a high-level abstraction (an abstract data type – ADT)
- Provides a convenient and effective mechanism for process synchronization
- It has
  - A name
  - Local variables
  - Local procedures (or functions)
  - Initialization section
  - Special variables called "condition" (condition C)
  - Only two operations on condition are allowed. If **C** is a condition variable, then:

```
1. c.wait();     // process is suspended until another process wakes it up.
2. c.signal();   // resumes exactly one suspended process.
                 // If no one is waiting on it, no effect
```

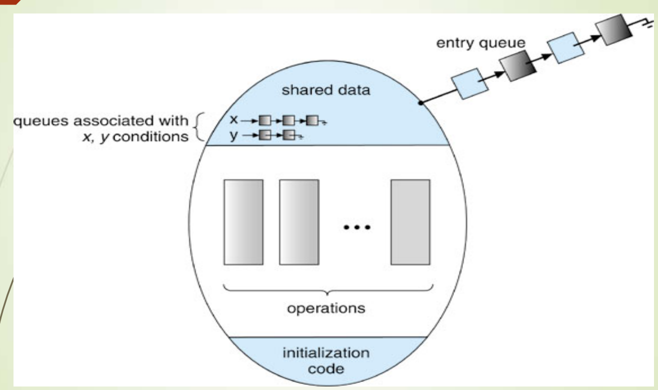

Monitors

- A programmer can define one or more variables of type *condition* as:
  `condition x, y;`
- Syntax of monitor type cannot be used directly by various processes.
- A procedure defined within monitor can access its local variables and formal parameters.
- Local variables of a monitor can be accessed by only the local procedures.

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (. . . ) { . . . }
  function P2 (. . . ) {. . . }
    . . .
  function Pn (. . . ) {. . . }

  Initialization code (. . . ) {. . . }
}
```


Schematic view of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

52

Condition Variables Choices

Design Issues:

- Let we have two processes P and Q such that:
  - P hits the `c.wait()` → blocks
  - Q enters monitor and issues the `c.signal()`
  - Now both P and Q are active (after signal) in the monitor
- Possible solution:
  - The signalling process Q is suspended until P (the wakening process) exits the monitor
  - The signalling process exits monitor only then Q is allowed to continue in the monitor
  - Force signal to be the last statement in a monitor code.
- Many languages have adopted the idea of monitors such as: Concurrent Pascal, Mesa, C#, Java

53

Producer-Consumer Problem using Monitors

```
Monitor ProduceConsumer{
   Condition full, empty;
   int count = 0;

   void enter(void){
    if (count == N)
      full.wait();
    enter_item;
    count++;
    if (count == 1)
      empty.signal();
   }

   void remove(void){
    if (count == 0)
      empty.wait();
    remove_item;
    count--;
    if (count == N - 1)
      full.signal();
   }
}
```

54

Producer-Consumer Problem using Monitors

```
void producer(void)
{
   while (true){
    produce_item();
    ProducerConsumer.enter();
   }
}

void consumer(void)
{
   while(true){
    ProducerConsumer.remove();
    consume_item();
   }
}
```

55

Monitor Solution to Dining Philosophers

- A deadlock-free solution to the dining-philosophers problem.
  - Solution imposes restriction that a philosopher may pick up her chopsticks only if both are available.
  - To code this solution, we need to distinguish among three states in which we may find a philosopher:

    `enum {thinking, hungry, eating} state[5];`
  - A philosopher i can set the variable `state[i] = eating` only if her two neighbors are not eating:

    `(state[(i+4)%5] != eating) and (state[(i+1)%5] != eating)`
  - We also need to declare: condition `self[5];`
  - where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.
- Distribution of chopsticks is controlled by the monitor `DiningPhilosophers`.
- Each philosopher, before starting to eat, must invoke the operation pickup() that may result in suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat.

Monitor Solution to Dining Philosophers

- Following this, the philosopher invokes the putdown() operation in the sequence:

```
DiningPhilosophers.pickup(i);
   ...
   eat
   ...
DiningPhilosophers.putdown(i);
```


Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
            self[i].wait;
}

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```


Solution to Dining Philosophers (Cont.)

```
void test (int i) {
  // Philosopher can set its own state EATING if both neighbors are not eating
        if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
             self[i].signal () ;
        }
}

    initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
     }
}      // for the monitor
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure *F* will be replaced by

```
wait(mutex);
     …
   body of F;
     …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

- Mutual exclusion within a monitor is ensured

60

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

61

Monitor Implementation (Cont.)

- The operation **`x.signal`** can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

62

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

63