Daniel **Fitz**
(43961229)

University Of Queensland

COMP3301 – Operating Systems Architecture

COMP3301 Lecture Notes

Table of Contents

List of Tables

List of Figures

Course Outline

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

Introduction

What is an operating system?

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

What Operating Systems Do

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

Operating System Definition

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

Common Functions of Interrupts

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

Interrupt Handling

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

Computer-System Architecture

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

Operating System Structure

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

Operating-System Operations

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

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

Transition from User to Kernel Mode

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

Process Management

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

Memory Management

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

Storage Management

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

IO Subsystem

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

Protection and Security

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

Open-Source Operating Systems

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

Operating-System Structures

Operating System Services

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

User Operating System Interface

CLI

CLI or command interpreter allows direct command entry

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

GUI

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

Touchscreen

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

System Calls

System Call Parameter Passing

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

Types of System Calls

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

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

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

System Programs

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

- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as services, subsystems, daemons

- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

Operating System Design and Implementation

- Design and Implementation of best OS not "solvable", but some approaches have proven successful
- User goals and System goals
  - **User goals** – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - **System goals** – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Important principle to separate

> Policy: What will be done?
> Mechanism: How to do it?

- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- Specifying and designing OS is highly creative task of software engineering

Implementation

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

UNIX

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

Layered Approach

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

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

Microkernel System Structure

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

Modules

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

System Boot

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

Processes

Process Concept

- An operating system executes a variety of programs
  - Batch system – jobs

Time-sharing systems – user programs or tasks

- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also call text section
  - Current activity including program counter, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is *passive* entity stored on disk (executable file), process is *active*

- Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

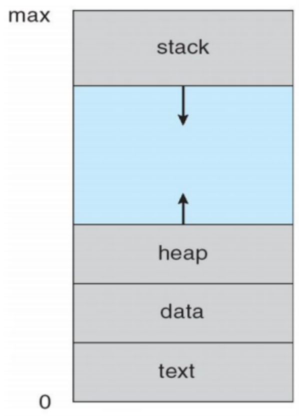

Figure 1: Process in Memory

Process State

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

![Diagram of Process State](sem2-2017/comp3301/state.png)$_{75}$

Process Control Block (PCB)

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

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

Threads

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

Process Scheduling

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

  - Processes migrate among the various queues

Schedulers

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

Addition of Medium Term Scheduling

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

Multitasking in Mobile Systems

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

Context Switch

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

Operations on Processes

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

Process Creation

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

    - Parent and children execute concurrently
    - Parent waits until children terminate
- Address space
    - Child duplicate of parent
    - Child has a program loaded into it
- UNIX examples
    - `fork()` system call creates new process
    - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

Process Termination

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

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

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Interprocess Communication

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

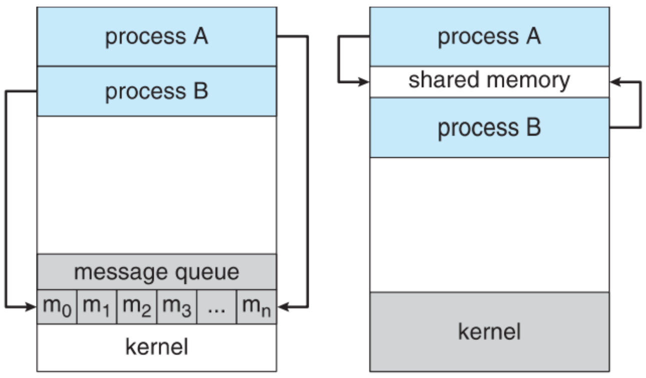

Figure 2: Communications Models

Cooperating Processes

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

Producer-Consumer Problem

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

Interprocess Communication - Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send(message)`** – message size fixed or variable
  - **`receive(message)`**
- If *P* and *Q* wish to communicate, they need:
  - establish a communication link between them
  - exchange message via send/receive
- Implementation of communication link
  - physical (e.g. shared memory, hardware bus)
  - logical (e.g. direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Direct Communication

- Processes must name each other explicitly:
  - **`send(P, message)`** – send a message to process P
  - **`receive(Q, message)`** – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - `send(A, message)` – send a message to mailbox A
  - `receive(A, message)` – receive a message from mailbox A

Synchronization

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

Buffer

- Queue of messages attached to the link; implemented in one of three ways

1) Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)

1) Bounded capacity – finite length of *n* messages
Sender must wait if link full

1) Unbounded capacity – infinite length
Sender never waits

Examples of IPC Systems

POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm fd, 4096);
```

  - Now the process could write to the shared memory

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

Communication in Client-Server Systems

Sockets

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

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters

- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - Big-endian and little-endian
- Remote communication has more failure scenarios than local
  - Messages can be delivered *exactly once* rather than *at most once*
- OS typically provides a rendezvous (or matchmaker) service to connect client and server

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

Windows calls these anonymous pipes

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Threads

Overview

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

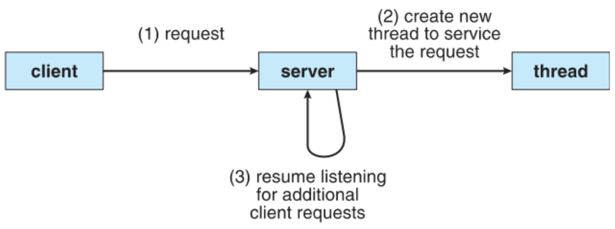
client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Figure 3: Multithreaded Server Architecture

Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- 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 number of threads grows, so does architectural support for threading
  - CPUs have cores as well as *hardware threads*

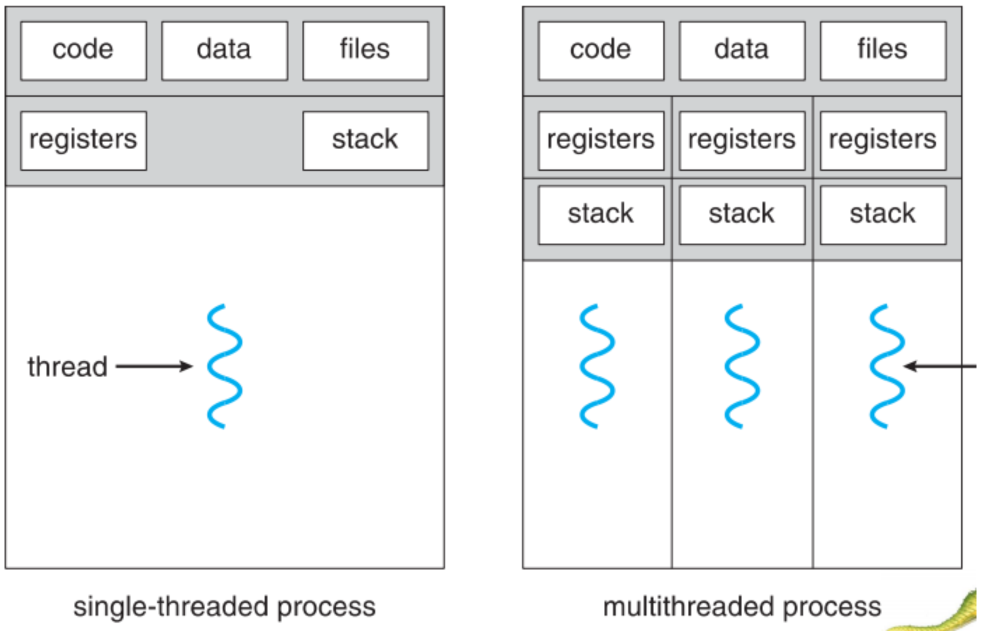

Figure 4: Single and Multithreaded Processes

Amdahls Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is a serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- i.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1/*S*

> Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

- **User threads** – management done by user-level threads library
- Three primary thread libraries
  1) POSIX pthreads
  2) Windows threads
  3) Java threads
- Kernel threads – supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

Multithreading Models

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 because only one may be in kernel at a time
- Few systems currently use this model

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

Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads

Thread Libraries

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

Pthreads

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

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:
  - Extending Thread class
  - Implementing the Runnable interface

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly 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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task

> i.e. Tasks could be scheduled to run periodically

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions – blocks of code that can run in parallel

Create as many threads as there are cores

```
#pragma omp parallel
```

Run for loop in parallel

```
#pragma omp parallel for
for (i=0; i<N; i++) {
c[i] = a[i] + b[i];
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in **`^{}`** – **`^{ printf("I am a block"); }`**
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - **Serial** – blocks removed in FIFO order, queue is per process, called main queue

        - Programmers can create additional serial queues within program
    - **Concurrent** – removed in FIFO order but several may be removed at a time
        - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{ printf("I am a block"); });
```

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Signal handling
    - Synchronous and asynchronous
- Thread cancellation of target thread
    - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
    - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
    1) Signal is generated by particular event
    2) Signal is delivered to a process
    3) Signal is handled by one of two signal handlers:
        1) default
        2) user-defined
- Every signal has **default handler** that kernel runs when handling signal
    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process
- Where should a signal be delivered for multi-threaded?
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process

Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e. when using a thread pool)
- Different from local variables
    - Local variables visible only during single function invocation
    - TLS visible across function invocations
- Similar to `static` data
    - TLS is unique to each thread

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)

- Flags control behavior

Flag	Meaning
`CLONE_FS`	File-system information is shared
`CLONE_VM`	The same memory space is shared
`CLONE_SIGHAND`	Signal handlers are shared
`CLONE_FILES`	The set of open files is shared

- `struct task_struct` points to process data structures (shared or unique)

CPU Scheduling

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU-I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

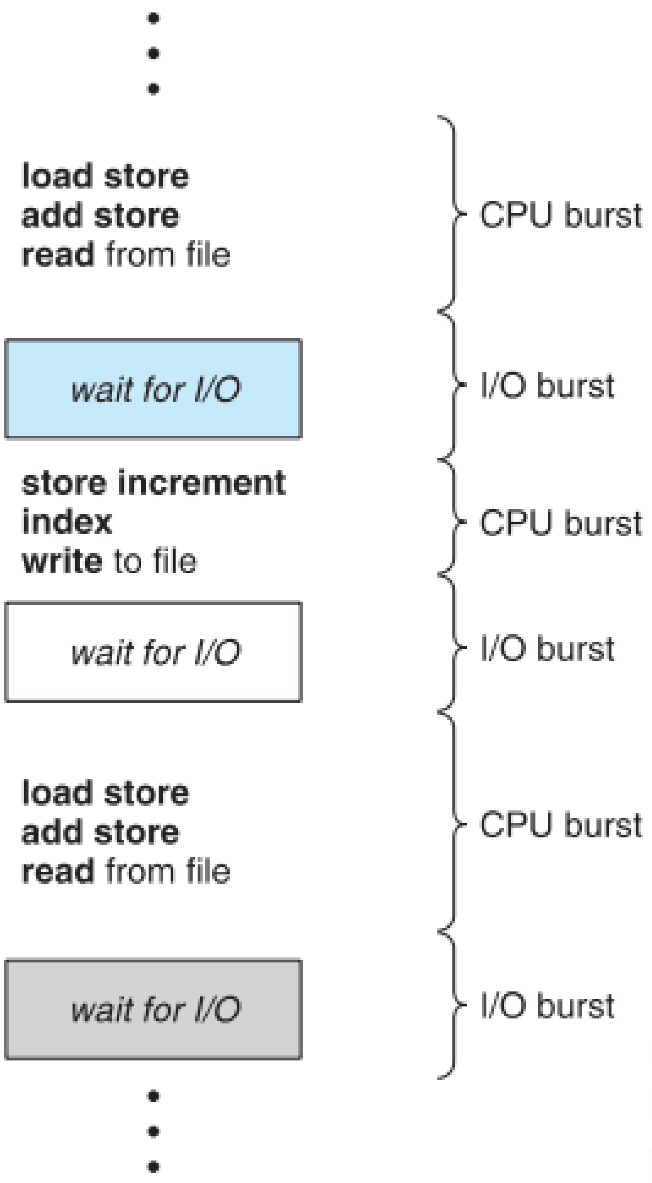

Figure 5: CPU Burst

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1) Switches from running to waiting state
  2) Switches from running to ready state
  3) Switches from waiting to ready
  4) Terminates
- Scheduling under 1 and 4 is nonpreemptive
- All other scheduling is preemptive
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, (for time-sharing environment)

Scheduling Algorithms

Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

> May be interested in AVERAGE or WORST CASE figures

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3. The Gantt Chart for the schedule is:

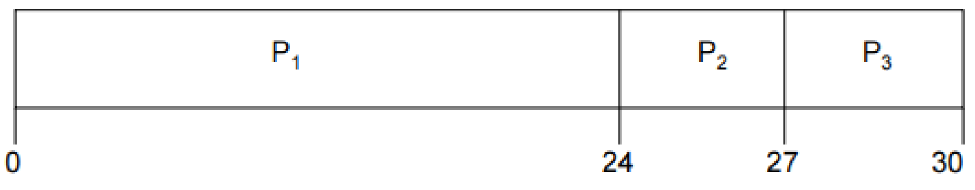

Figure 6: FCFS

- Average waiting time is 17

Suppose that the processess arrive in the order: P_2, P_3, P_1

- Average waiting time is 3
- Much better than previous case
- **Convoy effect** – short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU requent
  - Could ask the user

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
- Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
1. \mathcal$_{t}${n+1} = predicted value for the next CPU burst
1. \alpha, 0 \leq\alpha\leq 1
1. Define:

$$$
\mathcal{t}_{n=1} = \alpha t_n + (1 - \alpha) \mathcal{t}_n
$$$

- Commonly, \alpha set to 1/2
- Preemptive version called **shortest-remaining-time-first**

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

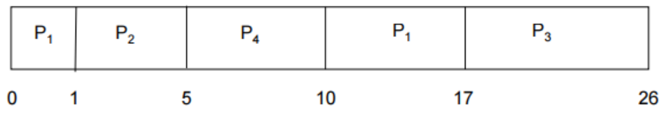

Figure 7: Preemptive SJF Gantt Chart

- Average waiting time = 6.5

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem = **Starvation** – low priority processes may never execute
- Solution = **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

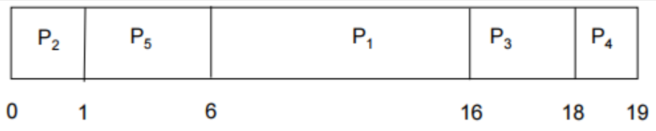

Figure 8: Priority Scheduling Example

• Average waiting time = 8.2

Round Robin

• Each process gets a small unity of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue

• If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets *1/n* of the CPU time in chunks of at most *q* time units at once. No process waits more than *(1-n)q* time units.

• Timer interrupts every quantum to schedule next process

• Performance

  • *q* large → FIFO

  • *q* small → *q* must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue

• Ready queue is partitioned into separate queues, e.g.

  • **foreground** (interactive)

  • **background** (batch)

• Process permanently in a given queue

• Each queue has its own scheduling algorithm:

  • foreground – RR

  • background – FCFS

• Scheduling must be done between the queues:

  • Fixed priority scheduling (i.e. serve all from foreground then from background). Possibility of starvation

  • Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e. 80% to foreground in RR

  • 20% to background in FCFS

Multilevel Feedback Queue

• A process can move between the various queue; aging can be implemented this way

• Multilevel-feedback-queue scheduler defined by the following parameters:

  • number of queues

  • scheduling algorithms for each queue (may be different)

  • method used to determine when to upgrade a process

  • method used to determine when to demote a process

  • method used to determine which queue a process will enter when that process needs service

• Most general algorithm, but also most complicated

Thread Scheduling

• Distinction between user-level and kernel-level threads

• When threads supported, threads scheduled, not processes

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP (lightweight processes)
    - Known as process-contention scope(**PCS**) since scheduling competition is within the process
    - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (**SCS**) – competition among all threads in system

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
    - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
    - soft affinity
    - hard affinity
    - Variations including processor sets

Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power than multiple chips
- Multiple threads per core also growing
    - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
    - Not knowing it doesn't own the CPUs
    - Can result in poor response time
    - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
    1) Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
    2) Dispatch latency – time for schedule to take current process off CPU and switch to another
- Conflict phase of dispatch latency:
    1) Preemption of any process running in kernel mode
    2) Release by low-priority process of resources

Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - 0 \leq *t* \leq *d* \leq *p*
  - Rate of periodic task is *1/p*

Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority
  - the later the deadline, the lower the priority

Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive *N/T* of the total processor time

Operating Systems Examples

Linux Scheduling

- Completely Fair Scheduler (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1) default
    2) real-time
- Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates target latency – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time
- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139

Windows Scheduling

- Windows uses priority-based preemptive scheduling

- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until:
  1) blocks
  2) uses time slice
  3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Solaris Scheduling

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modelling**
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms
  - Non-preemptive SFJ is 13ms
  - RR is 23ms

Queueing Models

- Describes the arrival of processes, and CPU and IO bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Littles Law

- *n* = average queue length
- *W* = average waiting time in queue
- *\lambda* = average arrival rate into queue
- Little's law - in steady state, processes leaving queue must equal processes arriving, thus *n=\lambda x W*
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

Process Synchronization

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data consistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

The Critical-Section Problem

- Consider system of *n* processes *{p_0, p_1, ... p{n-1}}*
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

General structure for a process:

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

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 a nonzero speed
  - No assumption concerning relative speed of the *n* processes

- Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Petersons Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `bool flag[2];`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process $\mathbf{p_i}$ is ready

Algorithm for Process Pi

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while (true);
```

- Provable that

1) Mutual exclusion is preserved
2) Progress requirement is satisfied
3) Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions

> Atomic = non-interruptible

  - Either test memory word and set value
  - Or swap contents of two memory words

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problems
- Simplest is mutex lock
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**
  - Can be OK for short waits on a multi-processor system

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify *S*: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**. The inventor of semaphores was Edsger Dijkstra who was very Dutch. **P** and **V** are the first letters of two Dutch words *proberen* (to test) and *verhogen* (to increment).
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- Can implement a counting semaphore *S* as a binary semaphore
- Can solve various synchronization problems
- Consider $\mathbf{P_1}$ and $\mathbf{P_2}$ that require $\mathbf{S_1}$ to happen before $\mathbf{S_2}$

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

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** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Deadlock and Starvation

- **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
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes

Bounded-Buffer Problem

- *n* buffers, 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*
- 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);
```

- The 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);
```

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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set

Readers-Writers Problem Variations

- First variation – no reader kept waiting unless writer has permission to use shared object
- Second variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbours, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore **`chopstick[5]`** initialized to 1

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** ... **`wait (mutex)`**
  - **`wait (mutex)`** ... **`wait (mutex)`**
  - Omitting of **`wait(mutex)`** or **`signal(mutex)`** (or both)
- Deadlock and starvation

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor t a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name {
    // shared variable declarations
    procedure P1 (...) {
        ...
    }

    procedure P2 (...) {
        ...
    }

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

Condition Variables

- `condition x, y;`
- Two operations 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

Condition Variables Choices

- If process *P* invokes `x.signal()`, with *Q* in `x.wait()` state, what should happen next?
  - If *Q* is resumed, then *P* must wait
- Options include
  - Signal and wait – *P* waits until *Q* leaves monitor or waits for another condition
  - Signal and continue – *Q* waits until *P* leaves the monitor or waits for another condition

  - Both have pros and cons – language implmenter can decide
  - Monitors implmented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Linux Synchronisation

- Linux:
  - Prior to kernel version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - atomic integers (set, add, sub, inc, read)
  - mutex locks
  - semaphores
  - spinlocks
  - reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption (i.e. disabling interrupts)

Atomic Transactions

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Must also work with multiple concurrent clients
- Approaches – see textbook for more detail if needed

- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

Deadlocks

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

System Model

- System consists of resources
- Resource types $R_1, R_2, ..., R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., $P\{n-1\}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc
- Task 1

```
Lock(Resource_1);
Lock(Resource_2);
// Do Stuff
Unlock(Resource_2);
Unlock(Resource_1);
```

- Task 2

```
Lock(Resource_2);
Lock(Resource_1);
// Do Stuff
Unlock(Resource_1);
Unlock(Resource_2);
```

Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*

- *V* is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$
- Can be analysed to detect deadlock

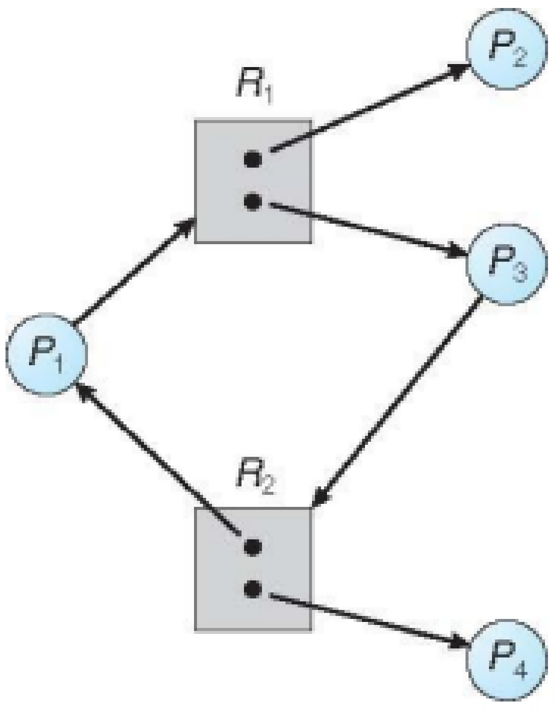

Figure 9: Resource-Allocation Graph

Basic Facts:

- If graph contains no cycles => no deadlock
- If graph contains a cycle =>
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the systeml used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence <P_1, P_2, ..., P_n> of ALL processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, retuen allocated resources, and terminate
  - When P_i terminates, *P{i+1}* can obtain its needed resources, and so on

Basic Facts:

- If a system is in safe state => no deadlocks
- If a system is in unsafe state => possibility of deadlock
- Avoidance => ensure that a system will never enter an unsafe state

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Bankers Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

> Not assessable for exam

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock

Recovery from Deadlock

Process Termination:

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1) Priority of the process
  2) How long process has computed, and how much longer to completion
  3) Resources that process has used
  4) Resources process needs to complete
  5) How many processes will need to be terminated
  6) Is process interactive or batch?

Resource Preemption:

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor