

CS241 Giga Cheat Sheet

14 October 2020 13:13

Welcome. Making this made me cry. I hope it helps you in your OSN adventures.

- Use the revision index to navigate to the question topic
- Click "notes" to see my personal notes on the topic made from the lectures
- Navigate to the page number specified to see the textbook section for that topic

Note: page numbers refer to the PDF page, not the textbook page, so use the same textbook.

[Operating System Concepts](#) // [Download](#)

[Computer Networking: A Top-Down Approach](#) // [Download](#)

Revision Index

Mock Papers

Module Overview

50% OS & 50% networks

- What is an OS, what are the main functions, how is it designed – how to analyse, write and design programs at an OS level
- In networks, learn communication between computers via networks, learn how networks are built, and how to use networks

Coursework is 20%

- Develop a multi-threaded intrusion detection system in C
- Nov 5th release, Dec 9th hand in

Exam is 80%

- April 2021
- online open book

[Moodle](#)

[Module Page](#)

[Jorel's Help](#)

Official OS Exam Revision

<https://www.os-book.com/OS10/review-dir/index.html>

Revision Index

02 April 2021 10:47

Operating Systems [Book](#)

Introduction to Operating Systems // [Mock Paper](#)

[Main services of the OS](#)
[Dual mode operation](#)
[Kernel design approaches](#)

Processes // [Mock Paper](#)

[Structure of a process in memory](#)
[Different states of a process](#)
[Process Control Block](#)
[Context Switching](#)
[Process creation and termination](#)
[Pros and cons of different models of inter-process communication](#)

Threads // [Mock Paper](#)

[Processes vs Threads](#)
[Concurrency vs Parallelism](#)
[Amdahl's Law](#)
[Different relations between user and kernel level threads](#)
[Multi-threaded server: 1 thread per request vs Thread pool](#)

Synchronisation // [Mock Paper](#)

[Race conditions](#)
[Critical section problems](#)
[Peterson's solution](#)
[Spinlocks, mutexlocks, semaphore](#)
[Deadlock, Starvation, Priority Inversion](#)
[Classical synchronisation problems](#)

Scheduling // [Mock Paper](#)

[CPU utilisation: Burst cycles](#)
[Different scheduling criteria](#)
[Different scheduling algorithms](#)

Deadlocks // [Mock Paper](#)

[Necessary conditions for a deadlock occurrence](#)
[Resource allocations graphs, deadlock detection](#)
[Deadlock prevention](#)

Main Memory // [Mock Paper](#)

[Contiguous memory allocation: External and internal fragmentation](#)
[Segmentation](#)
[Paging](#)

Networks [Book](#)

Introduction to Computer Networks // [Mock Paper](#)

[Delay computation - types of delay](#)
[Packet vs Circuit switching](#)
[Internet protocol stack](#)

Selected Topics In Networking // Mock Paper

[TCP Connection Establishment](#)
[SYN Attacks](#)
[MAC Addresses](#)
[ARP](#)

Application Layer // [Mock Paper](#)

[HTTP - persistent vs non-persistent](#)
[HTTP - cookies](#)
[HTTP - web caches](#)
[HTTP - conditional GET](#)

Transport Layer // [Mock Paper](#)

[TCP vs UDP](#)
[Reliable data transfer protocols](#)
[Stop and wait ARQ](#)
[Pipelined vs non-pipelined protocols](#)
[Go back N](#)
[Selective Repeat](#)
[TCP: Reliable data transfer](#)
[Flow control](#)
[Congestion control](#)

Network Layer // [Mock Paper](#)

[Longest prefix matching](#)
[Subnets](#)
[CIDR](#)
[NAT](#)
[Dijkstra's Algorithm](#)
[Distance vector algorithm](#)

Operating Systems

Introduction to Operating Systems

[questions](#)

Main services of the OS

[notes](#)

page 32

Intermediary between user and hardware

The main job of an OS is to allocate hardware resources to user processes, and control their execution.

I/O

[notes](#)

page 42

Interrupts

[notes](#)

page 36

Implementation on page 38

Event vector table on page 39

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed

location. Usually, the PCB of the process that is interrupted is stored and returned to once the interrupt has been serviced.

Storage

[notes](#)

page 40

There are various types of memory that are used by the OS, with varying sizes, speeds and persistence.

Dual mode operation

[notes](#)

page 52

Since the OS and its users share the hardware and software resources, an OS must ensure that an incorrect program cannot cause other programs to execute incorrectly.

Kernel design approaches

[notes](#)

page 111

Monolithic Structure page 112

Layered Approach page 113

Microkernels page 114

Modules page 116

There is no one perfect kernel design, so there are a variety in practice that take inspiration from different kernel design methodologies

Processes

[questions](#)

Structure of a process in memory

[notes](#)

page 144

There is a virtual address space of a process that is allocated, in which the following reside: text, data, stack, heap

Different states of a process

[notes](#)

page 146

New, running, waiting, ready, terminated

Process Control Block

[notes](#)

page 147

Each process is represented in the OS by a PCB. It contains information associated with a specific process.

Context switching

[notes](#)

page 152

Interrupts cause the OS to change a CPU core from its current task and to run a kernel routine.

Process creation and termination

[notes](#)

creation page 154

termination page 159

Pros and cons of different models of inter-process communication

[notes](#)

page 162

shared memory 163

message passing 165

Shared memory and message passing

Threads

[questions](#)

[notes](#)

page 216

Benefits of threads - page 219

Challenges in multicore systems - page 220

Data/Task parallelism - page 223

Threading issues - page 245

Processes vs Threads

[notes](#)

Thread creation has less overhead compared to creating entirely new processes

Concurrency vs Parallelism

[notes](#)

Parallelism is when multiple processes are operating at the same time on different cores, whereas concurrency is where a single core can appear to run multiple processes at the same time by rapidly switching between servicing two different processes.

Amdahl's Law

[notes](#)

The serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

page 221

Different relations between user and kernel level threads

[notes](#)

User level threads are implemented by the user and have no kernel involvement, thus lower overhead.

Kernel level threads are implemented in the kernel space and can be processed on different CPU's, but have more overhead

Multi-threaded server: 1 thread per request vs Thread pool

[notes](#)

page 223

In a threadpool model, a main thread creates a fixed number of threads that service requests. Less

overhead since threads only need creating at run time and never again.
One thread per request creates a new thread whenever one is needed, so can handle large volumes of burst, but has more overhead.

Synchronisation

[notes](#)

[questions](#)

page 332

Race conditions

[notes](#)

Critical section problem

[notes](#)

page 335

Peterson's solution

[notes](#)

page 337

Spinlocks, Mutexlocks, Semaphores

[notes](#)

mutex locks - page 346

semaphores - page 347

Deadlock, Starvation, Priority Inversion

[notes](#)

deadlock - page 358

priority inversion - page 359

Classical synchronisation problems (bounded buffer, readers-writers, dining philosophers)

[notes](#)

Bounded buffer problem - page 376

Solution to the first readers-writers synchronisation problem page 378

Dining philosophers problem - page 379

Scheduling

[notes](#)

[questions](#)

page 266

preemptive and non-preemptive scheduling - page 269

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

CPU utilisation: Burst cycles

[notes](#)

page 266

Different scheduling criteria

[notes](#)

page 271

Different scheduling algorithms

[notes](#)

page 272

First Come, First Served - page 273

Shortest Job First - page 274

Round Robin - page 276

Priority Scheduling - page 279

Deadlocks

[questions](#)

page 415

Deadlock handling - page 424

Necessary conditions for a deadlock occurrence

[notes](#)

page 419

- Mutual exclusion – only one process at a time can use a resource
- Hold and Wait – there must be a process holding some resources while waiting to acquire additional resources held by other processes
- No pre-emption – a resource can be released only voluntarily, and it cannot be snatched
- Circular wait – there must exist a subset of processes waiting for each other in a circular manner

Resource allocations graphs, deadlock detection

[notes](#)

page 421

Deadlock prevention

[notes](#)

page 425

If we can ensure one of the necessary conditions don't happen, we can avoid deadlocks.

Main Memory

[questions](#)

[notes](#)

page 455

Logical vs Physical address space page 459

Memory protection page 463

Memory protection in paged environments page 474

Contiguous memory allocation: External and internal fragmentation

[notes](#)

page 464

Main memory must accommodate the OS and the user processes. We therefore need to allocate main memory in the most efficient way possible. One method is CMA.

Segmentation

[notes](#)

The user's view of memory is not the same as the actual physical memory. Segmentation provides a way of mapping the programmer's view to the actual physical memory, so that the system has more freedom to manage memory.

Paging

[notes](#)

page 466

Shared pages - page 476

Structure of the page table - page 477

Inverted page tables 480

Segmentation permits the physical address space of a process to be non-contiguous. Paging is another memory management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

Networks

Introduction to Computer Networks

[questions](#)

Delay computation - types of delay

[notes](#)

page 62

Packets start at the host and pass through a series of routers, ending at the destination host. There are several types of delay that occur when the package is on this journey: nodal processing delay, queueing delay, transmission delay, and propagation delay, resulting in a total nodal delay.

Packet vs Circuit switching

[notes](#)

packet switching - page 50

circuit switching - page 54

In a network application, end systems exchange messages. The packets of these messages travel through communication links and packet switches (routers and link-layer switches). Packets are transmitted over each communication link at a rate equal to the full transmission rate of the link. So if a source end system or a packet switch is sending a packet of L bits over a link with transmission rate R bits/sec, then the time to transmit the packet is L/R .

Internet protocol stack

[notes](#)

page 77

Network designers organise protocols into layers. The IP protocol has 5 layers: application, transport, network, link, physical. The protocols of the various layers are called the protocol stack.

A layer can be implemented in software, in hardware, or in both. Application layer and transport layer protocols like HTTP and SMTP are almost always in software in end systems.

Physical and Link layers are typically implemented in a NIC associated with a given link. The network layer is often a mix.

Selected Topics In Networking

TCP Connection Establishment

[notes](#)

page 294

To establish a TCP connection from a client to a host, a 3 way handshake occurs, after which both client and host have buffers and variables allocated for each other and data transfer can begin.

SYN Attacks

page 299

SYN cookies - page 300

Since the server allocates resources to a client after the first initial SYN packet is received, a malicious actor can spam the server with SYN packets but never actually initiate a full connection, causing the server to allocate all its resources to fake connections that never end up actually being fulfilled. This is a SYN flood attack.

This can be combatted using SYN cookies.

MAC Addresses

[notes](#)

page 525

Each network interface has a globally unique identifier called a MAC address that is static and does not change regardless of where the card is or what network it is connected to.

ARP

[notes](#)

page 527

Determines MAC address using IP addresses. Uses broadcasting, and is susceptible to MITM attacks.

Application Layer

[questions](#)

HTTP - persistent vs non-persistent

[notes](#)

page 131

HTTP - cookies

[notes](#)

page 139

Cookies allow sites to keep track of users via an ID number. The site associates a connection with an ID number, that is stored in the users browser. Whenever that browser connects again, the site can load the users previous information (pages visited, items viewed, actions taken etc) without that user even having an account.

HTTP - web caches

[notes](#)

page 142

A web cache sits between the origin server and the client. When a client requests an object, it is cached in the web cache so that the next time that object is requested, it can be returned quickly from the web cache (that usually has a high bandwidth connection to the client) rather than having to be fetched from the origin server.

HTTP - conditional GET

[notes](#)

page 146

Web caches introduce the problem of stale content that has been updated on the origin server since it was last cached. Conditional GETs allow us to probe the origin server to see if a cached object has been modified since it was last fetched and stored. If so, the get will return the object. Otherwise, it will return 304 Not Modified.

Transport Layer

[questions](#)

TCP vs UDP

[notes](#)

TCP - page 272

TCP segment structure - page 275

TCP round trip time - page 280

TCP reliable data transfer - page 283

UDP - page 238

UDP segment structure - page 241

UDP checksum - page 241

Reliable data transfer protocols

[notes](#)

page 244

Building a reliable data transfer protocol - page 247

Handling bit errors in ARQ protocols - page 248

Checksums, ACKs, Timeouts, Retransmissions (Automatic Repeat Request (ARQ)) and Sequence Numbers

Stop and wait ARQ

[notes](#)

page 248

Automatic Repeat reQuest protocols are reliable data transfer protocols based on retransmissions.

Pipelined vs non-pipelined protocols

[notes](#)

page 256

Consequences of pipelining - page 260

Pipelined protocols allow senders to send multiple packets without waiting for acknowledgements.

Go back N

[notes](#)

page 260

timeout, receipt, invocation - page 263

The sender is allowed to transmit multiple packets without waiting for ack, but is constrained to have no more than some max number N of unacked packets in the pipeline

Selective Repeat

[notes](#)

page 265

GBN allows the sender to potentially fill the pipeline with packets, thus avoiding the channel utilisation problems we noted with Stop-And-Wait. However, there are scenarios where GBN has performance issues, relating to when window size and bandwidth delay product are both large and there are many packets in the pipeline. A single packet error can result in many packets being retransmitted.

Selected Repeat avoids unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error.

TCP: Reliable Data Transfer Protocols

[notes](#)

page 284

IP is unreliable. TCP creates a reliable data transfer service on top of IP's unreliable best-effort service. TCP's RDTS ensures that the data stream is uncorrupted, without gaps, without duplication and in sequence.

TCP: Flow Control

[notes](#)

page 291

If an application is slow at reading the data sent to it, the sender can easily overflow the connections receive buffer by sending too much data too quickly. TCP Provides flow control to its applications. It is a speed matching service, matching the rate at which the sender is sending against the rate the receiver is reading.

TCP: Congestion Control

[notes](#)

page 302

Causes and costs of congestion - page 302

Approaches to congestion control - page 309

TCP congestion control - page 311

Slow start - page 313

A sender can be throttled to handle congestion on a network. Congestion control mechanisms handle this throttling to reduce congestion.

Network Layer

[questions](#)

Longest prefix matching

[notes](#)

page 362

When there are multiple matches, the router uses the longest prefix matching rule; that is, it finds the longest matching entry in the table and forwards the packet to the link interface associated with the longest prefix match

Subnets

[notes](#)

interface addresses and subnets - page 382

Subnet addresses - page 383

A subnet or subnetwork is a smaller network inside a large network. Subnetting makes network routing much more efficient.

CIDR

[notes](#)

page 383

The Internet's address assignment strategy is known as Classless Interdomain Routing.

CIDR generalizes the notion of subnet addressing. As with subnet addressing, the 32-bit IP address is divided into two parts and again has the dotted-decimal form a.b.c.d/x, where x indicates the number of bits in the first part of the address

NAT

[notes](#)

page 392

Network Address Translation allows for the use of private IP addresses in a private subnet. These IP addresses only have meaning within the private network. When packets are forwarded beyond the home network in the global internet, they can no longer use these private IP's, as a source or destination address.

A NAT enabled router does not look like a router to the outside world. It looks like a single device with an IP address. All traffic leaving the home router has that IP address, and all traffic entering the home router must have that destination address.

When datagrams arrive at the NAT from the WAN, the port number is used to lookup the local IP address in the NAT translation table.

Dijkstra's Algorithm

[notes](#)

page 428

Computes the least cost path from the source node to destination node

Distance vector algorithm

[notes](#)

page 433

The link state algorithm (dijkstra's) uses global information, whereas the distance vector algorithm is iterative, asynchronous, and distributed.

- It is distributed in the sense that each node receives some information from one or more of its directly attached neighbours, performs a calculation, then distributes the results of its calculation back to its neighbours.
- It is iterative in that this process continues on until no more info is exchanged
- It is asynchronous in that it does not require all of the nodes to operate in lockstep with each other.

It is related to the Bellman-Ford equation from dynamic programming

Mock Tests

02 April 2021 11:42

Intro to Operation Systems
Processes
Threads
Synchronisation
Scheduling
Deadlocks
Main Memory

Intro to Networks
Selected Topics in Networking
Application Layer
Transport Layer
Network Layer

Intro to Operating Systems

What would be the major issues in a computer system if the OS did not manage I/O operations

Consider an old OS that does not support dual mode operation. Explain a major drawback of the OS

State one advantage and one disadvantage of the layered approach to kernel design

What are the purpose of system calls?

What is the purpose of the command interpreter? why is it usually separate from the kernel?

What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

What is the purpose of system programs?

What is the main advantage of the microkernel approach to system design? What are the main disadvantages?

List five services provided by an operating system and explain how each creates convenience for the users. In which cases would it be impossible for user level programs to provide these services? Explain

Why do some systems store the OS in firmware, while others store it on disk?

How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Processes

What are the two models of interprocess communication? What are the strengths and weaknesses of both?

Briefly describe the different states of a process and the transition among them

A process is represented in the OS by a PCB - describe a scenario when the OS uses the PCB

3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process? a. Stack b. Heap c. Shared memory segments

What is a zombie process?

Describe the actions taken by a kernel to context-switch between processes.

Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level. a. Synchronous and asynchronous communication b. Automatic and explicit buffering c. Send by copy and send by reference d. Fixed-sized and variable-sized messages

In UNIX, orphan processes

A. Do not exist

B. Execute normally and terminate as orphan processes

C. Are assigned a parent

D. Become zombies

What is TRUE?

A. Shared memory communication has less overhead than message passing

B. Shared memory communication uses kernel help to send and receive messages

C. In shared memory systems, producer and consumer processes are automatically synchronised

D. None

What is TRUE for Unix?

A. A pipe can only be established between a parent and a child

B. Pipe exists even after the process creating the pipe terminates

C. Named pipes can be bi-directional

D. A and C

E. A, B, and C

How many processes are created?

A. 2

B. 3

C. 4

D. 8

E. 16

first loops

main process forks

now we have two processes

second loop:

two processes fork, creating 4 processes with i = 3

final loop:

all processes fork, creating 8 total

Threads

Can a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single process system Briefly explain your answer

INTRO TO OS ANSWERS

The user would not be able to interrupt the CPU with I/O requests.

Protection - only one process can access at a time

Dual-mode operation allows for differentiation between kernel and user processes. Allowing user processes to modify the kernel would be a critical security flaw and could result in the OS being damaged.

Layered kernel design is easier to modify because functionality is split into layers, allowing only a single layer to be updated at a time. This also makes debugging easier as it allows bugs to be traced to contained areas of the kernel. However, lower layers can never access the services of higher layers, making it difficult to split the kernel into definitive layers. This also leads to duplicate code between layers and adds overhead to system calls, since they need to traverse layers.

System calls are low level operations, including memory management and file writing. They allow USER PROGRAMS to make privileged calls.

skip 5

skip 6

System programs are collections of system calls, that allow for kernel functionality to be grouped together. This makes interaction between user programs and the kernel cleaner and shorter.

Microkernel approach: a stripped down kernel that moves all non-essential functionality to the user space. This is advantageous because it results in a very small kernel, which is much more secure and reliable. However, it means that many switches between user and kernel mode are needed, resulting in more overhead.

- 1) Memory management - allows users read/write access to directories and files
- 2) I/O management - allows the user to input information into their computer using peripherals
- 3) Handles failures and errors - manages and avoids fatal errors, or situations that would result in undefined behaviour e.g. two processes writing to the same memory location twice
- 4) Handles interprocess communication - allows processes to communicate with each other via message passing / shared memory, which extends the functionality of the computer for the user.

PROCESSES ANSWERS

- 1) Message passing - relies on a send and receive operation that is implemented using system calls. The kernel is relied on to make sure that the messages are received. Can be implemented with direct or indirect communication between processes, however indirect (use of a mailbox with an ID - receiver reads from mailbox) is preferred. This is like Shared memory but the mailbox is managed by the kernel
Pros: easier to implement on distributed systems
No conflict avoidance needed as managed by kernel, so useful for small bits of information
Cons: Repeated system calls cause overhead.

Shared memory - a shared space that the processes can put messages into. A process is a producer or consumer at any moment.

Pros:

Less overhead because the only system calls are the initial memory allocation

Can be faster due to lack of need for kernel

Cons:

Need to manage conflicts where two producers try to write at same time

- 2) New - process is going created - goes to:
ready - process is waiting to be assigned to a processor - goes to:
running - process instructions are actively being executed - goes to:
terminated
back to ready
waiting - waiting for an I/O event etc - goes to ready again
- 3) When an interrupt signal is received, the OS must store the PCB of the currently executing process, so that it can transition away to service the interrupt, then transition back to the process in its exact state, by loading the PCB
- 4) Concurrent processing can introduce race conditions
The instantiation of large numbers of concurrent processes, and the necessary co-ordination can cause significant overhead, potentially causing the system to OOM
Concurrent processing can lead to starvation of processes
- 5) m
- 6) A zombie process is one which has finished executing, but still exists in the process table because it's parent process has not yet received the child's exit status.
- 7) Context switching - when interrupts cause the OS To change a CPU core from its current task to run a kernel routine.
First, the kernel saves the current state into the PCB.
Secondly, the kernel loads the interrupting process into the CPU using its PCB.
Once the interrupt has been serviced, the state of the interrupt process is saved into its PCB, and the PCB of the first process is loaded back into memory
- 8) Named pipes persist after they have been established, so would be more useful for processing large numbers of jobs that have an unknown execution time e.g. connecting to a website via HTTP
Regular pipes are good for directly processing data from one process to another, e.g. when piping the result of one process into the input of another

INTRO TO OS

What would be the major issues in a computer system if the OS did not manage I/O operations

User would not be able to interact with their machine unless they implement I/O operations themselves

Consider an old OS that does not support dual mode operation. Explain a major drawback of the OS

Major security concern as multiple programs can write to the same device potentially whipping out the OS

State one advantage and one disadvantage of the layered approach to kernel design

What are the purpose of system calls?

Calls provide basic functionality to the user to operate the OS, such as process control, file management and information maintenance

What is the purpose of the command interpreter? why is it usually separate from the kernel?

What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

- The fork() system call creates a new process
- The exec() system call is then used to load the program into memory

What is the purpose of system programs?

- System programs – a collection of many system calls
- Purpose – provide user with a convenient environment for development and execution of programs

What is the main advantage of the layered approach to system design? What are the main disadvantages?

- Layered design – OS is broken down into multiple layers, hardware being at the bottom and user interface being at the top
- Advantages – Modular approach allows for easier debugging
- Disadvantages – More overhead as data cannot skip layers

List five services provided by an operating system and explain how each creates convenience for the users. In which cases would it be impossible for user level programs to provide these services? Explain

- Program execution – OS allows user to execute programs by providing a convenient environment
- I/O operations – OS provides an environment to handle I/O operations of programs, e.g. an output file, input device such as a keyboard
- File system management – OS provides environment for CRUD operations on files
- Process communication – OS takes care of communication between processes, e.g. sending data packets to a network device
- Error detection – OS constantly monitors the systems and all running processes

Why do some systems store the OS in firmware, while others store it on disk?

- It makes more sense to embed the OS in the firmware for devices that lack I/O ports, e.g. game console, smart speakers...
- I/O ports such that a disk can be read with the appropriate OS

How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

- Requires a boot manager program that must be stored on the hard drive so that it is recognized during system start up
- Boot manager then allows user to boot into different OS that must be stored on separate disks

Threads

Can a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single process system Briefly explain your answer

Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single processor system?

50% of a programs execution time is improved by a factor of 4. The remaining 50% is improved by a factor of 2. Calculate the speedup of the program

In a multi-threaded server, why is threadpool better than one thread per request?

Where are mutex locks used in the threadpool strategy

Where are the condition variables used in the threadpool strategy

Provide three programming examples in which multithreading provides better performance than a single threaded application

Provide three examples in which multithreading provides no benefit over a single threaded application

What are the differences between user-level threads and kernel-level threads, and under what circumstances would you use one over the other?

Describe the actions taken by a kernel to context-switch between kernel level threads

What resources are used when a thread is created? How do they differ from the resources needed when a process is created?

Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Which of the following components of program state are shared across threads in a multithreaded process? a. Register values b. Heap memory c. Global variables d. Stack memory

Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

Is it possible to have concurrency but not parallelism? Explain.

Using Amdahl's Law, calculate the speedup gain for the following applications: • 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores • 67 percent parallel with (a) two processing cores and (b) four processing cores • 90 percent parallel with (a) four processing cores and (b) eight processing cores

In multi-threading, multiple threads share which of the following?

A. Code, Files, and Data
B. Code, Files, Data, and Heap
C. Code, Stack, Data
D. Code, Register, Data

Is the following loop parallelizable?

```
A. Yes          for (i=0; i < ARR_SIZE-1; i++)  
B. No           j[i]=1+log2(i);  
C. Partially    }
```

1/5 of a program's execution time is improved by a factor of 4 by parallelizing. What is the speed up obtained?

A. 5/8
B. 2
C. 4/5
D. 1/2

When a thread is blocked on an OS system call, the entire process is blocked. What type of mapping is being used between user and kernel threads?

A. One-to-one
B. Many-to-one
C. Many-to-Many
D. Many-to-one

Synchronisation

Why are semaphores considered more powerful synchronisation primitives than mutex locks?

Why are spinlocks not appropriate for single-processor systems yet are often used in multiprocessor systems

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems

The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism — a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock

Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

Consider the code example for allocating and releasing processes shown in Figure 6.20. a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable int number of processes = 0 with the atomic integer atomic t number of processes = 0 to prevent the race condition(s)?

```
while (true) {  
    while (true) {  
        flag[i] = want_in;  
        j = turn;  
  
        while (j != i) {  
            if (flag[j] != idle) {  
                j = turn;  
            }  
            else  
                j = (j + 1) % n;  
        }  
  
        flag[i] = in_cs;  
        j = 0;  
  
        while ( (j < n) && (j == i || flag[j] != in_cs) )  
            j++;  
  
        if ( (j >= n) && (turn == i || flag[turn] == idle) )  
            break;  
    }  
  
    /* critical section */  
  
    j = (turn + 1) % n;  
}
```

THREADS ANSWERS

- 2) Yes. A multithreaded system can provide better performance because it allows for parallelisable tasks to be completed in parallel rather than consecutively. A single process system does not have this capability
- 3) No, the OS only sees a single process and will not schedule the different threads of the process on separate processes, so no performance benefit
- 4) Speedup of a factor of 2.667
- 5) Threadpool model has the benefit of reserve capacity - threads do not need to be instantiated when a request is made as a reserve pool of threads is created once at the start of execution, which persist over time and so can process jobs instantly. Further, by setting a limit of the thread number, the system can't OOM as easily as a one-thread-per-request model that OOM during a burst of activity
- 6) Around the critical section
- 7) Condition variables are used around the section of the code where the thread queries for a job to process. If there are no jobs, the thread will wait() on the condition variable, until another thread signals the variable with signal(), causing the waiting thread to wake up and query for a job.
- 8) Multithreading examples:
Processing raw network data due to its bursty nature
A web scraping application, since web scraping can be intensive, one thread per website would allow for multiple sites to be processed simultaneously
Video games - different objects in a game can be implemented as different threads, e.g. a car, an NPC etc
- 9) Multi-threading can be a bad idea in applications that require precise physical time
Examples?
User level - implemented in the user space, and the kernel is not aware of them. The is no kernel involvement in a user level thread, so there is less overhead. They cannot run in parallel on different CPU's.
Kernel level - implemented by the kernel in the kernel space, and therefore can be scheduled on different CPU's. However, they are kernel managed, so there is more overhead.
Applications create lightweight user level threads, but they must ultimately be mapped to kernel level threads to execute on a CPU. This is a many-to-one relationship.
- When to use each?
- 11) Thread context switching - keep memory cache, same code, global vars same,
- 12) A thread needs an ID, a program counter, a register and a stack. A process, on the other hand, needs global data and code
- 13) (If first is not limited to one core) then when there are > 1 cores in a multithreaded solution using kernel threads, the kernel threads can take advantage of the other cores to gain an advantage over the single processor system
- 14)

SYNCHRONISATION ANSWERS

- 1) Mutex locks are binary values - locked or not locked. A semaphore has an integer value and therefore can be used to solve many different synchronisation problems
- 2) A spinlock "spins" indefinitely at a condition in the code, constantly checking to see if the condition is the correct value before it proceeds. This means that the CPU is constantly occupied, even though no useful computation is occurring. In multiprocessor systems this is not an issue because computation can still occur in other cores. It is beneficial in that the computation waiting at the spinlock can begin instantly when the lock is freed, without having to perform a context switch.
- 3) Case 1 -- interrupts are disabled for ONE processor only -- result is that threads running on other processors could ignore the synchronization primitive and access the shared data

Case 2 -- interrupts are disabled for ALL processors -- this means task dispatching, handling I/O completion, etc. is also disabled on ALL processors, so threads running on all CPUs can grind to a halt
- 4) Use conditional variable that is signalled when there is a change to the resource associated with the mutex lock
- 5) Spinlock since no context switch is required which may take more time than the actual wait. Other processors can continue to compute while the spinlock is waiting on one core
Mutex lock since the sleeping thread will allow the CPU to perform some useful computation during the long wait period
Mutex lock since it is possible that the thread may get put to sleep anyway, so we may as well try to perform some useful computation while waiting
- 6) Upper bound for holding a spinlock would be 2T
-
7) We could use a counting semaphore initialised with the maximum number of sockets that we wish to have open at any one time.
When a socket requests to be initialised, we can perform a wait() operation on the semaphore, thereby decrementing the count. Upon termination of a socket connection, we can signal() the semaphore to decrement the count. If the semaphore decrements to 0, any socket connection that wishes to be initialised will be blocked, until another socket terminates and increments the counter
- 8) We could implement a priority inheritance protocol. Where all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When the process is finished, the priority would revert to their original values.

```

while (true) {
    while (true) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else {
                j = (j + 1) % n;
            }
        }

        flag[i] = in_cs;
        j = 0;

        while ( (j < n) && (j == i || flag[j] != in_cs) )
            j++;

        if ( (j == n) && (turn == i || flag[turn] == idle) )
            break;
    }

    /* critical section */

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    /* remainder section */
}

```

Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Scheduling

What is "time quantum" and why do we use it? What is the downside of having a small time quantum?

Explain the difference between pre-emptive and Nonpreemptive scheduling.

What's a round-robin scheduler, and what are the benefits over a priority scheduler?

A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n .

Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use Nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process / Arrival Time / Burst Time

P1 0.0 8

P2 0.4 4

P3 1.0 1

- What is the average turnaround time for these processes with the FCFS scheduling algorithm?
- What is the average turnaround time for these processes with the SJF scheduling algorithm?
- The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the CPU Scheduling average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.

The following processes are being scheduled using a pre-emptive, round robin scheduling algorithm.

Process / Priority / Burst / Arrival

P1 40 20 0

P2 30 25 25

P3 30 25 30

P4 35 15 60

P5 5 10 100

P6 10 10 105

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an idle task (which consumes no CPU resources and is identified as *Pidle*). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is pre-empted by a higher-priority process, the pre-empted process is placed at the end of the queue.

- Show the scheduling order of the processes using a Gantt chart.
- What is the turnaround time for each process?
- What is the waiting time for each process?
- What is the CPU utilization rate?

What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on. These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

- Priority and SJF
- Multilevel feedback queues and FCFS
- Priority and FCFS
- RR and SJF

Of these two types of programs:

- I/O-bound
- CPU-bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer

Discuss how the following pairs of scheduling criteria conflict in certain settings.

- CPU utilization and response time
- Average turnaround time and maximum waiting time
- I/O device utilization and CPU utilization

Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

- $\alpha = 0$ and $\tau_0 = 100$ milliseconds
- $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

A variation of the round-robin scheduler is the regressive round-robin scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated to the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

Which of the following scheduling algorithms could result in starvation?

- First-come, first-served
- Shortest job first
- Round robin
- Priority

sockets that we wish to have open at any one time.

When a socket requests to be initialised, we can perform a wait() operation on the semaphore, thereby decrementing the count. Upon termination of a socket connection, we can signal() the semaphore to decrement the count. If the semaphore decrements to 0, any socket connection that wishes to be initialised will be blocked, until another socket terminates and increments the counter

- We could implement a priority inheritance protocol. Where all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When the process is finished, the priority would revert to their original values.

SCHEDULING ANSWERS

- Time quantum is a length of time assigned to a process by a scheduler. The smaller the time quantum, the more often the CPU has to context switch, reducing the amount of useful computation that a computer can do in a given time period.
- In non-pre-emptive scheduling, the CPU is given up voluntarily after a process has finished executing or when the process switches to the waiting state
In pre-emptive scheduling, the CPU is "snatched" mid-execution. This can cause problems for processes that share data, in that if the CPU is snatched mid-way through the updating of data, then that data may be in an inconsistent state when the next process tries to access it.
- Round-robin scheduler is a scheduler that allocates a small time quantum to every process. After the time has elapsed, the process is pre-empted and added to the end of the ready queue. Usually, the scheduler visits the processes in order of their arrivals
e.g. if there are N processes in the ready queue, then each process gets $1/N$ CPU time at most, in chunks of q at once.
- non-pre-emptive? $n! \times (N/(q \times n))$
pre-emptive? $n! \times (N/(q \times n))$
e.g. if we have $N=9$ seconds, and time quantum $q=1$ sec, with $n=3$ processes
Then there are $3! \times (9/(1 \times 3)) = 216$ different schedules

p1	p2	p3	p3	p1	p2	p1	p3	p2
----	----	----	----	----	----	----	----	----

- part 1)

8

$(8+4) \cdot 0.4 = 11.6$

$(8+4+1) \cdot 1 = 12$

$8 + 11.6 + 12 = 31.6$

$31.6 / 3 = 10.5333$

- part 2)

8

$(1+8) \cdot 1 = 8$

$9+4 \cdot 0.4 = 12.6$

$8+8+12.6 = 28.6$

$28.6 / 3 = 9.5333$

- part 3)

idle for 1

1 (turnaround for first job) (since it arrives at 1, gets processed by 2)

$0.6+1+4 = 5.6$ (turnaround for 2, since it arrives at 0.4, idles until 2, then takes 4)

$1 + 1 + 4 + 8 = 14$ (idles 1, waits 1 for first job, 4 for second job, 8 for itself)

$1 + 5.6 + 14 = 20.6$

$20.6 / 3 = 6.8667$

- utilisation time = 120 (total time) 15 (wasted idle time) 105 (productive compute)
- Typically because in a multilevel queue, the processes with the highest priority are those that are interactive. They need to have a good response time, and as such a smaller time quantum. CPU bound processes on the other hand (tend to be lower priority) require raw CPU time and as such don't benefit from more frequent context switches.
- Priority and Shortest Job First**
A shortest job first algorithm is just a priority algorithm in which the priority number is determined by the job length.
Multilevel feedback queues and FCFS
None
Priority and FCFS
A FCFS queue can be built with a priority queue where all jobs get priority 0
RR and SJF
You can make a SJF scheduler using a RR with a very small time quantum (probably not this)

- I/O bound programs are more likely to have voluntary context switches, since they tend to block for additional I/O input, and thus give up the CPU voluntarily while waiting for I/O. Further, they tend to be interactive and thus higher priority than CPU batch jobs, making them more likely to take priority over other processes that may want to preempt

CPU-bound processes tend to exhaust all available CPU time given to them, only giving up the CPU when they are pre-empted

- CPU utilisation conflicts with response time in RR schedulers, since a fast response time requires shorter time quanta and thus more context switches. The increased number of context switches causes the CPU utilisation to drop, since time spent context switching cannot be used for any productive compute.

Average turnaround time conflicts with maximum waiting time ?????

I/O Device Utilisation conflicts with CPU utilisation for the same reasons as specified in the first conflict.

- CPU bound since they don't block for I/O as frequently
- xxx
- Starvation can occur in priority schedulers, since the CPU may end up constantly processing jobs of a higher priority, and never have time to process a very low priority job.
This can be solved by aging.
- 2 pointers to the same process = two time quanta in a row of computation
???

Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

- What would be the effect of putting two pointers to the same process in the ready queue?
- What would be two major advantages and two disadvantages of this scheme?
- How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Consider a pre-emptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α . When it is running, its priority changes at a rate β . All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

- What is the algorithm that results from $\beta > \alpha > 0$?
- What is the algorithm that results from $\alpha < \beta < 0$?

Explain the how the following scheduling algorithms discriminate either in favour of or against short processes:

- FCFS
- RR
- Multilevel feedback queues

What are the possible values of x after both processes finish executing?

- 1
- {-1,0,1,2}
- {-1,1,2}
- {-1,2}
- None

```
//shared variable
int x=0;

P1:
x=x+2;

P2:
x=x-1;
```

Which of the three criteria of the CS problem will this solution satisfy?

- Mutual exclusion
- Progress
- Bounded waiting
- A, B, C
- A, B
- None

```
//shared variable
boolean flag[2]={false,false}

P0:
flag[0]=true;
while (flag[1])
{
//critical section
flag[0]=false;
//remainder section
}

P1:
flag[1]=true;
while (flag[0])
{
//critical section
flag[1]=false;
//remainder section
}
```

What is the maximum possible value of x after the execution of all processes?

- 2
- 0
- 1
- 2
- None

```
//shared variable
int x=0;
semaphore S=1;

P1:
wait(S);
x=x+1;
signal(S);

P2:
wait(S);
x=x+1;
signal(S);

P3:
wait(S);
x=x-2;
signal(S);
```

Vevox: 148 - 800 - 179

What is the maximum possible value of x after the execution of all processes?

- 2
- 0
- 1
- 2
- None

```
//shared variable
int x=0;
semaphore S=2;

P1:
wait(S);
x=x+1;
signal(S);

P2:
wait(S);
x=x+1;
signal(S);

P3:
wait(S);
x=x-2;
signal(S);
```

Vevox: 148 - 800 - 179

Which of the following process scheduling algorithm may lead to starvation?

- FCFS
- SJF
- RR
- None

If the time quantum of round robin algorithm is very large, then it is equivalent to:

- FCFS
- SJF
- LCFS
- LJF

The order in which the processes complete under the FCFS and RR (with q=2) are

- FCFS: P1, P2, P3
RR: P1, P2, P3
- FCFS: P1, P3, P2
RR: P3, P1, P2
- FCFS: P1, P2, P3
RR: P3, P1, P2
- FCFS: P1, P2, P3
RR: P1, P3, P2

Process	Arrival Times	CPU time
P1	0	5
P2	1	7
P3	3	4

Deadlocks

Consider a system consisting of three resources of the same type that are shared by three processes, each of which needs at most two resources. Can a deadlock occur?

What is the optimistic assumption made in the deadlock detection algorithm? How can this assumption be violated?

What is starvation and how can it be avoided?

Can a system detect that some of its threads are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

Suppose that a system is in an unsafe state. Show that it is possible for the threads to complete their execution without entering a deadlocked state.

DEADLOCK ANSWERS

1

p1 has r1 and needs r2
p2 has r2 and need r3
p3 has r3 and needs r1

2

We optimistically assume that if a thread has requested some resources, and they are available, then that thread will use these resources and require no more to complete its task. It can be violated if the thread does in fact need more resources

3

Starvation occurs when a process has to wait indefinitely while other processes are making ..

5 - The BTV operating system has a 2²¹-bit virtual address, yet on certain emulated devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?
a. A conventional, single-level page table
b. An inverted page table
What is the maximum amount of physical memory in the BTV operating system?

9 - Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
a. How many bits are required in the logical address?
b. How many bits are required in the physical address?

10 - Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?
a. A conventional, single-level page table
b. An inverted page table

11 - Explain the difference between internal and external fragmentation

12 - Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

13 - Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

14 - Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
a. Contiguous memory allocation
b. Paging

15 - Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:
a. External fragmentation
b. Internal fragmentation
c. Ability to share code across processes

16 - On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

17 - Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)? a. 21205 b. 164250 c. 121357 d. 16479315 e. 27253187

18 - Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.
a. How many bits are required in the logical address?
b. How many bits are required in the physical address?

19 - Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following? a. A conventional, single-level page table b. An inverted page table

20 - What is the purpose of paging the page tables?

21 - Consider a paging system with the page table stored in memory.
a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?
b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

Introduction to Computer Networks

Why do computers use port numbers?

What is the difference between a host and an end system? List several different types of end systems. Is a Web server an end system?

R22. List five tasks that a layer can perform. Is it possible that one (or more) of these tasks could be performed by two (or more) layers?

R18. How long does it take a packet of length 1,000 bytes to propagate over a link of distance 2,500 km, propagation speed m/s, and transmission rate 2 Mbps? More generally, how long does it take a packet of length L to propagate over a link of distance d, propagation speed s, and transmission rate R bps? Does this delay depend on packet length? Does this delay depend on transmission rate?

R19. Suppose Host A wants to send a large file to Host B. The path from Host A to Host B has three links, of rates R1=500 kbps, R2=2 Mbps, and R3=1 Mbps. Assuming no other traffic in the network, what is the throughput for the file transfer? b. Suppose the file is 4 million bytes. Dividing the file size by the throughput, roughly how long will it take to transfer the file to Host B? c. Repeat (a) and (b), but now with R reduced to 100 kbps.

R20. Suppose end system A wants to send a large file to end system B. At a very high level, describe how end system A creates packets from the file. When one of these packets arrives to a router, what information in the packet does the router use to determine the link onto which the packet is forwarded? Why is packet switching in the Internet analogous to driving from one city to another and asking directions along the way?

P2. Equation 1.1 (delay end to end = NLR) gives a formula for the end-to-end delay of sending one packet of length L over N links of transmission rate R. Generalize this formula for sending P such packets back-to-back over the N links.

P13. a. Suppose N packets arrive simultaneously to a link at which no packets are currently being transmitted or queued. Each packet is of length L and the link has transmission rate R. What is the average queuing delay for the N packets? b. Now suppose that N such packets arrive to the link every LN/R seconds. What is the average queuing delay of a packet?

P10. Consider a packet of length L that begins at end system A and travels over three links to a destination end system. These three links are connected by two packet switches. Let d₁, s₁, and R denote the length, propagation speed, and the transmission rate of link 1, for . The packet switch delays each packet by d₂. Assuming no queuing delays, in terms of d₁, s₁, R, and L, what is the total end-to-end delay for the packet? Suppose now the packet is 1,500 bytes, the propagation speed on all three links is the transmission rates of all three links are 2 Mbps, the packet switch processing delay is 3 msec, the length of the first link is 5,000 km, the length of the second link is 4,000 km, and the length of the last link is 1,000 km. For these values, what is the end-to-end delay?

P24. Suppose you would like to urgently deliver 40 terabytes data from Boston to Los Angeles. You have available a 100 Mbps dedicated link for data transfer. Would you prefer to transmit the data via this link or instead use FedEx over-night delivery? Explain

What will be the delay in sending a packet of L bits from A to B?

A. L/R
B. 2L/R
C. 3L/R
D. 4L/R

What will be the delay in sending two packets of L/2 bits each from A to B without any gap between their transmissions?

A. L/R
B. 2L/R
C. 3L/R
D. 4L/R

$$\log_2(32 \cdot 1024) = 13$$

5

6

memory	l	ff	bf	wf
300kb		115		
600kb		500		500
350kb		200		200
200kb				200
750kb		358		358 115
125kb				115

processes:

- 115kb
- 500kb
- 358kb
- 200kb
- 375kb

7

$$3085/1024 = 3.0117$$

Page number = 3
Offset = 13

$$42095/1024 = 41.1084$$

page = 41
offset = 111

$$215201/1024 = 210.1572$$

page 210
offset 161

$$650000/1024 = 634.7656$$

page 634
offset = 784

$$2000001/1024 = 1953.126$$

page = 1953
offset = 129

8

2^21

9

Logical address space of 256 pages, 4kb page size.

Physical memory of 64 frames.

20 bits for logical address

18 bits for physical address

10

32 bit logical address and 4kb page size

512mb physical memory

a) $2^{32} / (1024 \cdot 4) = 1048576 = 2^{20}$

b) $512 \cdot 1024 / 4 = 131072 = 2^{17}$

11

External fragmentation = when there exists enough memory to satisfy a request, but it is not contiguous as it is spread across many small sections. It occurs as processes are loaded into, and removed from memory. Around 50% of memory is lost to fragmentation and may be unusable, in best fit systems

Internal fragmentation = unused memory that is internal to a partition.

12

13

14

15

16

17

trivial

18

19

20

INTRO TO COMPUTER NETWORKS ANSWERS

- Computers use port numbers to identify processes with a system
- A host and an end system mean the same thing. They are interchangeable terms for a device on the edge of a network, like a phone, computer, or laptop.
End systems include some computers with which the end user does not interact, like mail servers and web servers.
- Layers are responsible for generating, transmitting and receiving data that is sent over the Internet.
Application layer - generates the data that can be communicated over the Internet (HTTP, SMTP)
Transport Layer - Packetizes the data with port number, adds sequencing and error correcting info (TCP, UDP)
Network Layer - Add source and dest IP addresses, and runs routing algorithms (IP, routing protocols)
Link Layer - Add source and dest MAC (Ethernet)
Physical Layer - send the individual packets through the physical communication layer
- Yes, two layers can do the same thing e.g. network and transport layer both do flow control
Generally speaking.
Propagation delay = link distance / propagation speed
Bandwidth delay (or transmission delay) = packet size / transmission rate
In our case, PD = 2500 x 1000 / (2.5x10^8) = 0.01 , or 10ms
BD = 1000 / ((2048*1024)/8) = 0.0038s = 3.8ms
- Bottleneck link is R1, so 500kbps
62.5 sec
R2, 312.5
- Creating packets from a file:
Application layer - generates the message
Transport layer - Packetize the message and add initial headers (e.g. port number)
Network layer - Add source and destination IP address and run the routing algorithm
Link layer - Add source and destination MAC addresses
Physical layer - send individual bits through the communication layers
Reverse this process at the destination

The router uses the IP address of the packet to determine where to send it to. This is looked up in

- B. 2L/R
C. 3L/R
D. 4L/R



What is the maximum achievable throughput between A and B?

- A. 10bps
B. 8bps
C. 12bps
D. 30bps



What is TRUE in the Internet?

- A. Packets between the same src-dest pair can take different routes
B. Packets between two different src-dest pairs cannot share same links
C. Packets cannot arrive out of order
D. Packets can be lost
E. A, B, D
F. A, D

Which layer(s) is(are) present in all network nodes?

- A. Application layer
B. Transport layer
C. Network layer
D. Link layer
E. Physical layer
F. All five
G. Link and Physical Layers
H. Network, Link, and Physical Layers

Which layer(s) is(are) present ONLY on the end hosts?

- A. Application layer
B. Transport layer
C. Network layer
D. Link layer
E. Physical layer
F. All 5 Layers
G. Application, Transport, and Network Layers
H. Application and Transport Layers

What is FALSE for TCP?

- A. TCP guarantees the delivery of all packets sent
B. TCP guarantees orderly delivery of all packets
C. TCP guarantees a constant throughput to applications
D. TCP provides connection-oriented service
E. TCP has more overhead than UDP

Transport layer - Packetize the message and add initial headers (e.g. port number)

Network layer - Add source and destination IP address and run the routing algorithm

Link layer - Add source and destination MAC addresses

Physical layer - send individual bits through the communication layers

Reverse this process at the destination

The router uses the IP address of the packet to determine where to send it to. This is looked up in the routing table of the router. The table keeps IP prefixes since many IP's will match to the same outgoing link. Longest prefix matching is used to determine which of the existing links to send the packet to

- 7) End to end delay = NL/R
If we have P packets, the delay is $N+P \cdot 1(L/R)$
Since there are first P delays, getting each packet to the first link one -by-one. Then, there are $N-1$ delays getting the packets over the last link
8) First packet has no queuing delay, second has L/R , third $2L/R$ etc
Average delay is $(N-1)L / 2R$ (sum of natural numbers)
It takes $N \cdot L/R$ time to process a set. If the sets arrive every $L / R \cdot N$ seconds then we can process the first set before the second arrives
9) Packet L , beings at A
3 Links connected by two switches
 d = length
 s = speed
 R = trans rate

APPLICATION LAYER ANSWERS

- The port number at the end of the IP address
- UDP, since TCP's lossless transmission results in slower transfer speeds
- An example of such an application would be a stock trading system, that needs to communicate accurate information at high-speed.
In this case I would use TCP with a persistent connection, because although it is slower, it guarantees delivery
- A transport protocol provides the following 4 services:
Reliable data transfer
Throughput - both UDP and TCP have constant throughput
Timing - retransmit on timeout - TCP yes, UDP no
Security -

In order delivery - TCP yes, UDP no - TCP numbers packets
Resending - TCP yes, UDP no - packets are resent if they are corrupted or missing
Reliable data transfer: UDP no, TCP yes
Flow control - ensures large numbers of packets are not sent to the receiver at the same time: UDP no, TCP yes
- Handshaking - the process by which a connection can be initiated between two a client and a server. In the case of TCP, this involves a three-step process in which the client sends a small TCP segment to the server, the server acknowledges, and then the client sends the HTTP request combined with the third part of the handshake
- These protocols run on TCP because they require the application data to be received without any gaps, in the correct order. TCP can provide this because it is a reliable, connection-oriented service.
- Susan connects to amazon.com
• Server creates a unique ID number and creates an entry in its database, indexed by ID number
The Amazon web server then responds to Susan's browser, including in the HTTP response a set cookie header which contains the ID number: set-cookie: 1678 •
• When Susan's browser receives the HTTP response message, it sees set-cookie
• The browser then appends a line to the special cookie file that it manages. • This line includes the hostname of the server and the ID number in the set-cookie header • Note that the cookie file already has an entry for other sites that Susan uses. As Susan uses Amazon, each time she requests a page, her browser consults the cookie file, extracts her ID, and puts a cookie header, and puts a cookie header line that includes the ID number in the HTTP request • Specifically, each of her HTTP requests to the Amazon server include cookie: 1678 • Amazon is able to track Susan's activity at the site • Although the server does not know Susan's name, it knows exactly what pages user 1678 has Example of Cookies in Action CS241 OS and Networks Page 87 Although the server does not know Susan's name, it knows exactly what pages user 1678 has visited, in which order, and when. • Amazon uses cookies to provide its shopping cart service - maintaining a list of all the items that Susan has added, despite her not being logged in. • When Susan signs up for Amazon, they can then associate an account with the cookie number, and fill it with all the information she requested in the past
- A web cache sits between a client and an origin server. When a client makes a request, the request is routed through the cache. If the entity exists already on the cache server, it can be returned directly to the client without having to request the entity from the origin. If the entity does not exist, then it still has to be retrieved and return to the client, however a copy is then stored in the cache so that it doesn't have to be retrieved again when it is next requested. This will only speed up the requests for some entities that have been requested before.
- One end of the TCP connection has to be connected to the client, and the other to a server socket. This is because TCP is a connection oriented protocol that requires a connection to be established before data transfer can commence. UDP has no such requirement and as such just needs to connect to the client socket.
A TCP server supporting n simultaneous connections would need $n+1$ sockets (one welcome socket, one outgoing socket per connection)
- a) false b) true c) false d) false e) false
- 2RTT0
- a) 2RTT to get the main file, then 2RTT for each of the 8 files, so 18RTT
- b) 2RTT to get the main file, then 2RTT to get 5, 2RTT to get the last 3, so 6RTT
- c) 2RTT??

TRANSPORT LAYER ANSWERS

R1)

segment size 1200 bytes,
dest host from transport later

Network guarantees to deliver the segment to the transport layer at the dest host
Many network app processes can be running at dest
4 Byte port number

Requirements:

Needs to be reliable like TCP

Needs to have a unique port number - destination address from Transport layer

Application Layer

R5. What information is used by a process running on one host to identify a process running on another host?

R6. Suppose you wanted to do a transaction from a remote client to a server as fast as possible. Would you use UDP or TCP? Why?

Can you conceive of an application that requires no data loss and that is also highly time -sensitive? What protocol would you use for this particular application, and why?

R8. List the four broad classes of services that a transport protocol can provide. For each of the service classes, indicate if either UDP or TCP (or both) provides such a service.

R10. What is meant by a handshaking protocol?

R11. Why do HTTP, SMTP, and POP3 run on top of TCP rather than on UDP?

R12. Consider an e-commerce site that wants to keep a purchase record for each of its customers. Describe how this can be done with cookies.

R13. Describe how Web caching can reduce the delay in receiving a requested object. Will Web caching reduce the delay for all objects requested by a user or for only some of the objects? Why?

R26. In Section 2.7, the UDP server described needed only one socket, whereas the TCP server needed two sockets. Why? If the TCP server were to support n simultaneous connections, each from a different client host, how many sockets would the TCP server need?

P1. True or false?

- A user requests a Web page that consists of some text and three images. For this page, the client will send one request message and receive four response messages.
- Two distinct Web pages (for example, www.mit.edu/research.html) and www.mit.edu/students.html) can be sent over the same persistent connection.
- With nonpersistent connections between browser and origin server, it is possible for a single TCP segment to carry two distinct HTTP request messages.
- The Date: header in the HTTP response message indicates when the object in the response was last modified.
- HTTP response messages never have an empty message body

P7. Suppose within your Web browser you click on a link to obtain a Web page. The IP address for the associated URL is not cached in your local host, so a DNS lookup is necessary to obtain the IP address. Suppose that n DNS servers are visited before your host receives the IP address from DNS; the successive visits incur an RTT of R . Further suppose that the Web page associated with the link contains exactly one object, consisting of a small amount of HTML text. Let RTT denote the RTT between the local host and the server containing the object. Assuming zero transmission time of the object, how much time elapses from when the client clicks on the link until the client receives the object?

P8. Referring to Problem P7, suppose the HTML file references eight very small objects on the same server. Neglecting transmission times, how much time elapses with

- Non-persistent HTTP with no parallel TCP connections?
- Non-persistent HTTP with the browser configured for 5 parallel connections?
- Persistent HTTP?

Transport Layer

R1. Suppose the network layer provides the following service. The network layer in the source host accepts a segment of maximum size 1,200 bytes and a destination host address from the transport layer. The network layer then guarantees to deliver the segment to the transport layer at the destination host. Suppose many network application processes can be running at the destination host.

- Design the simplest possible transport-layer protocol that will get application data to the desired process at the destination host. Assume the operating system in the destination host has assigned a 4-byte port number to each running application process.
- Modify this protocol so that it provides a "return address" to the destination process.
- In your protocols, does the transport layer "have to do anything" in the core of the computer network?

R2. Consider a planet where everyone belongs to a family of six, every family lives in its own house, each house has a unique address, and each person in a given house has a unique name. Suppose this planet has a mail service that delivers letters from source house to destination

- b. Modify this protocol so that it provides a "return address" to the destination process.
 c. In your protocols, does the transport layer "have to do anything" in the core of the computer network?

R2. Consider a planet where everyone belongs to a family of six, every family lives in its own house, each house has a unique address, and each person in a given house has a unique name. Suppose this planet has a mail service that delivers letters from source house to destination house. The mail service requires that
 (1) the letter be in an envelope, and that
 (2) the address of the destination house (and nothing more) be clearly written on the envelope. Suppose each family has a delegate family member who collects and distributes letters for the other family members. The letters do not necessarily provide any indication of the recipients of the letters.
 a. Using the solution to Problem R1 above as inspiration, describe a protocol that the delegates can use to deliver letters from a sending family member to a receiving family member.
 b. In your protocol, does the mail service ever have to open the envelope and examine the letter in order to provide its service?

R3. Consider a TCP connection between Host A and Host B. Suppose that the TCP segments traveling from Host A to Host B have source port number x and destination port number y. What are the source and destination port numbers for the segments traveling from Host B to Host A?

R4. Describe why an application developer might choose to run an application over UDP rather than TCP.

R5. Why is it that voice and video traffic is often sent over TCP rather than UDP in today's Internet? (Hint: The answer we are looking for has nothing to do with TCP's congestion-control mechanism.)

R6. Is it possible for an application to enjoy reliable data transfer even when the application runs over UDP? If so, how?

R7. Suppose a process in Host C has a UDP socket with port number 6789. Suppose both Host A and Host B each send a UDP segment to Host C with destination port number 6789. Will both of these segments be directed to the same socket at Host C? If so, how will the process at Host C know that these two segments originated from two different hosts?

R8. Suppose that a Web server runs in Host C on port 80. Suppose this Web server uses persistent connections, and is currently receiving requests from two different Hosts, A and B. Are all of the requests being sent through the same socket at Host C? If they are being passed through different sockets, do both of the sockets have port 80? Discuss and explain.

R15. Suppose Host A sends two TCP segments back to Host B over a TCP connection. The first segment has sequence number 90; the second has sequence number 110.
 a. How much data is in the first segment?
 b. Suppose that the first segment is lost but the second segment arrives at B. In the acknowledgment that Host B sends to Host A, what will be the acknowledgment number?

R17. Suppose two TCP connections are present over some bottleneck link of rate R bps. Both connections have a huge file to send (in the same direction over the bottleneck link). The transmissions of the files start at the same time. What transmission rate would TCP like to give to each of the connections?

R18. True or false? Consider congestion control in TCP. When the timer expires at the sender, the value of ssthresh is set to one half of its previous value

P1. Suppose Client A initiates a Telnet session with Server S. At about the same time, Client B also initiates a Telnet session with Server S. Provide possible source and destination port numbers for
 a. The segments sent from A to S.
 b. The segments sent from B to S.
 c. The segments sent from S to A.
 d. The segments sent from S to B.
 e. If A and B are different hosts, is it possible that the source port number in the segments from A to S is the same as that from B to S?
 f. How about if they are the same host?

P3. UDP and TCP use 1s complement for their checksums. Suppose you have the following three 8-bit bytes: 01010011, 01100110, 01101000. What is the 1s complement of the sum of these 8-bit bytes? (Note that although UDP and TCP use 16-bit words in computing the checksum, for this problem you are being asked to consider 8-bit sums.) Show all work.

Why is it that UDP takes the 1s complement of the sum; that is, why not just use the sum? With the 1s complement scheme, how does the receiver detect errors? Is it possible that a 1-bit error will go undetected? How about a 2-bit error?

P4.
 a. Suppose you have the following 2 bytes: 01011100 and 01100101. What is the 1s complement of the sum of these 2 bytes?

b. Suppose you have the following 2 bytes: 11011010 and 01100101. What is the 1s complement of the sum of these 2 bytes?

c. For the bytes in part (a), give an example where one bit is flipped in each of the 2 bytes and yet the 1s complement doesn't change.

P5. Suppose that the UDP receiver computes the Internet checksum for the received UDP segment and finds that it matches the value carried in the checksum field. Can the receiver be absolutely certain that no bit errors have occurred? Explain.

P23. Consider the GBN and SR protocols. Suppose the sequence number space is of size k. What is the largest allowable sender window that will avoid the occurrence of problems such as that in Figure 3.27 for each of these protocols?

P24. Answer true or false to the following questions and briefly justify your answer:
 a. With the SR protocol, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
 b. With GBN, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
 c. The alternating-bit protocol is the same as the SR protocol with a sender and receiver window size of 1.
 d. The alternating-bit protocol is the same as the GBN protocol with a sender and receiver window size of 1.

P25. We have said that an application may choose UDP for a transport protocol because UDP offers finer application control (than TCP) of what data is sent in a segment and when.
 a. Why does an application have more control of what data is sent in a segment?
 b. Why does an application have more control on when the segment is sent?

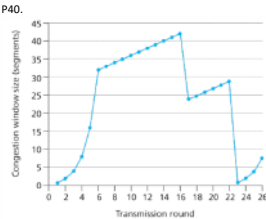


Figure 3.58 TCP window size as a function of time

- a. Identify the intervals of time when TCP slow start is operating.
 b. Identify the intervals of time when TCP congestion avoidance is operating.
 c. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
 d. After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
 e. What is the initial value of ssthresh at the first transmission round?
 f. What is the value of ssthresh at the 18th transmission round?
 g. What is the value of ssthresh at the 24th transmission round?
 h. During what transmission round is the 70th segment sent?
 i. Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of ssthresh?
 j. Suppose TCP Tahoe is used (instead of TCP Reno), and assume that triple duplicate ACKs are

Network guarantees to deliver the segment to the transport layer at the dest host
 Many network app processes can be running at dest
 4 Byte port number

Requirements:
 Needs to be reliable like TCP
 Network layer only accepts 1200 bytes + destination address from Transport layer
 Need to use port numbers

Transport Layer protocol:
 Packetize the data into packets of 1200 bytes including headers
 Headers should include:
 - source port (4 bytes)
 - destination port (4 bytes)
 - sequence number (for reliability) - 4 bytes
 - acknowledgement number (for reliability) - 4 bytes
 - checksum (for reliability) - 4 bytes
 - flags - SYN and ACK - 2 bits

In order for the protocol to be reliable, it should be connection-oriented, so a 3 way handshake should be initiated before sending any data

To ensure the data transferred is reliable, the checksum should be computed every time a packet is sent and received. If the checksums do not match, the data should be retransmitted. Further, a timeout should be set after the initial handshake. If sent data is not acknowledged within the timeout, it should be retransmitted.

R2)

When the delegate receives an outgoing letter, take note of the name of the destination person. The letter should then be put in one of six piles at the destination, each of which is assigned to a person in the house.

R3)

The same but switched around (A source is B dest and vice versa)

R4)

We would use UDP over TCP in the case that the arrival of all the data is not paramount, but rather the speed at which it arrives. E.g. if data is irrelevant/unusable after a certain time period (e.g. live video game data) then it is more important that we try to get the data to the destination as fast as possible, rather than ensuring that it all arrives correctly (since if it doesn't arrive on time, it is no longer usable anyway)

R5)

The prevalence of high-speed broadband means that the benefits of lossless transmission outweigh the benefits of faster, less reliable transmission

R6)

Yes, by implementing some TCP features.
 flow control - we'd need flow control to stop the receiver buffer overflowing and data being lost
 retransmission - foreign hosts should acknowledge packet received. If not, retransmission should occur.
 sequencing - packets should be ordered using a sequence number.

R7)

Yes. Both will be directed to the process assigned to port 6789. UPD headers have a source port and a destination port. The receiver can check the source ports and compare them.

R8)

TCP connections are determined by a 4-tuple consisting of the source and dest ports and IP addresses. If there are two different machines connecting to the web server on port 80, they will have two distinct connections because the source IPs will differ. If the same machine connects to the third machine on port 80, with two different connections, it will still be differentiated because of the port number.

R15)
 "The sequence number for a segment is the byte-stream number of the first byte in the segment"
 "The acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B - TCP only acknowledges bytes up to the first missing byte in the stream"

20 bytes in the first segment. Byte 90 will be acknowledged, because TCP only acknowledges bytes up to the first missing byte in the stream.

R17)

As per the TCP congestion control algorithm, the send rates will first be dictated by a slow start phase. This means that the initial sending rate will be set to 1MSS, resulting in a throughput of MSS/RTT for both of the connections

R18)

True. The sender then enters a slow start phase

P1)

A port = 1
 B port = 2
 S port = 3

a) source: 1, dest: 3
 b) source: 2, dest: 3
 c) source: 3, dest: 1
 d) source: 3, dest: 2
 e) Yes, it is possible. However, the IP addresses will be different, so they will still be able to be differentiated
 f) no - if they are the same host, with the same source port, then the connection is by definition the same

P3)

01010011
 +
 01100110

 =
 10111001
 +
 01110100
 =
 100101101

wrap the extra bit and add it:
 00101101
 1
 00101110

ones comp:
 11010001

Ones comp can be calculated by inverting the digits of the numbers

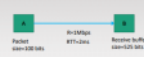
received at the 16th round. What are the ssthresh and the congestion window size at the 19th round?
k. Again suppose TCP Tahoe is used, and there is a timeout event at 22nd round. How many packets have been sent out from 17th round till 22nd round, inclusive?

The transport layer protocols used for real time multimedia, file transfer, DNS and email, respectively are:

A. UDP, UDP, TCP, TCP
B. UDP, TCP, UDP, TCP
C. TCP, UDP, TCP, UDP
D. UDP, TCP, TCP, UDP

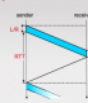
What is the maximum link utilisation achievable through a window-based pipelined protocol?

A. 1
B. 1/2
C. 1/4
D. 1/8
E. None




What is the maximum link utilisation achievable through a window-based pipelined protocol?

Maximum link utilisation,
$$U = \frac{\min(B, L + R \times RTT)}{L + R \times RTT}$$
$$= \min\left(1, \frac{B}{L + R \times RTT}\right)$$



What should be the window size to fully utilise the link if A uses GBN protocol?

A. 5
B. 11
C. 21
D. 51
E. 101



What is NOT TRUE?

A. SR protocol causes less retransmissions
B. GBN uses cumulative ACKs
C. An SR receiver cannot receive a packet with sequence number outside its receive window
D. In GBN, the receiver discards out of order transmissions

Assume that A and B use the SR protocol to communicate with window size N. Let m and n be smallest sequence numbers in the send and receive windows, respectively. What is TRUE?

A. m=n
B. m >= n
C. m < n
D. m <= n
E. B, D
F. A, C

Assume that A and B use the SR protocol to communicate with window size N. Let m and n be smallest sequence numbers in the send and receive windows, respectively. What is TRUE?

The sender cannot send packets beyond sequence number m+1
The receiver must have correctly received and acknowledged sequence number m
The receiver cannot receive and therefore ACK packets with the sequence number n

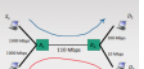
Answers:
a) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000

Suppose the a sender-receiver pair uses the SR protocol and a range of sequence numbers 0-7. What is the maximum window size they can use?

A. 7
B. 8
C. 4
D. 10

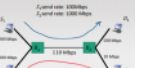
What are the maximum possible throughputs of the flow (S₁, D₁) and (S₂, D₂)?

A. 100 Mbps, 10 Mbps
B. 110 Mbps, 110 Mbps
C. 1000 Mbps, 1000 Mbps
D. 10 Mbps, 10 Mbps



What is the throughput of the flow (S₁, D₁)?

A. 100 Mbps
B. 110 Mbps
C. 10 Mbps
D. 1000 Mbps




TCP cwnd starts at 2MSS in the slow start phase with ssthresh=8MSS. What would be the value of cwnd at the 5th transmission round? Assume no loss detected

A. 32 MSS
B. 8 MSS
C. 10 MSS
D. 16 MSS

TCP cwnd starts at 2MSS in the slow start phase with ssthresh=8MSS. What would be the value of cwnd at the 5th transmission round? Assume no loss detected

Window size for 1st transmission = 2 MSS
Window size for 2nd transmission = 4 MSS
Window size for 3rd transmission = 8 MSS
threshold reached, increase linearly (according to AIMD)
Window size for 4th transmission = 9 MSS
Window size for 5th transmission = 10 MSS



Network Layer

- R1. Let's review some of the terminology used in this textbook. Recall that the name of a transport-layer packet is segment and that the name of a link-layer packet is frame. What is the name of a network-layer packet? Recall that both routers and link-layer switches are called packet switches. What is the fundamental difference between a router and link-layer switch?
- R3. We made a distinction between the forwarding function and the routing function performed in the network layer. What are the key differences between routing and forwarding?
- R4. What is the role of the forwarding table within a router?

P7. Consider a datagram network using 8-bit host addresses. Suppose a router uses longest prefix matching and has the following forwarding table:

Prefix Match	Interface
1	0
10	1
111	2
otherwise	3

For each of the four interfaces, give the associated range of destination host addresses and the number of addresses in the range.

P8. Consider a router that interconnects three subnets: Subnet 1, Subnet 2, and Subnet 3. Suppose all of the interfaces in each of these three subnets are required to have the prefix **223.1.17/24**. Also suppose that Subnet 1 is required to support at least 60 interfaces, Subnet 2 is to support at least 90 interfaces, and Subnet 3 is to support at least 12 interfaces. Provide three network addresses (of the form a.b.c.d/x) that satisfy these constraints.

P11. Consider a subnet with prefix 128.119.40.128/26. Give an example of one IP address (of form xxx.xxx.xxx.xxx) that can be assigned to this network. Suppose an ISP owns the block of addresses of the form 128.119.40.64/26. Suppose it wants to create four subnets from this block, with each block having the same number of IP addresses. What are the prefixes (of form a.b.c.d/x) for the four subnets?

P17. Suppose you are interested in detecting the number of hosts behind a NAT. You observe that the IP layer stamps an identification number sequentially on each IP packet. The identification number of the first IP packet generated by a host is a random number, and the identification numbers of the subsequent IP packets are sequentially assigned. Assume all IP packets generated by hosts behind the NAT are sent to the outside world.

- a. Based on this observation, and assuming you can sniff all packets sent by the NAT to the outside, can you outline a simple technique that detects the number of unique hosts behind a NAT? Justify your answer.
- b. If the identification numbers are not sequentially assigned but randomly assigned, would your technique work? Justify your answer.

PROCESSES

What are the two models of interprocess communication? What are the strengths and weaknesses of both?

- Message parsing - repeated system calls cause overhead but useful for passing smaller amounts of data
- Shared memory - less overhead because fewer system calls, however danger of critical section problem

Briefly describe the different states of a process and the transition among them

- New - process has just been created
- Ready - a "new" process is admitted to the ready to be executed state
- Running - scheduler picks a process from ready state and dispatches it
- Waiting - a running process can be halted, transitions into waiting state
- Terminated - a running process has terminated

A process is represented in the OS by a PCB - describe a scenario when the OS uses the PCB

- Context switching

3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing.

Discuss three major complications that concurrent processing adds to an operating system.

- Context switching requires system calls that may overload the system
- Risk of deadlocks when processes require big data blocks for execution
- Main memory might be heavily taxed as system needs to keep track of all processes

When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process? a. Stack b. Heap c. Shared memory segments

- Shared memory segments – stack and heap are not shared but copied to the new process

What is a zombie process?

- A process that has finished execution but still has an entry point in the process table

Describe the actions taken by a kernel to context-switch between processes.

- Receives system call or clock interrupt call – stores current process running on CPU in PCB
- Loads context of different process from its PCB into main memory

Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

- Establish communication between two specific process on the same machine - Ordinary pipe more suitable as it requires less overhead
- Communication over a network - Named pipes more suitable as they can be used to listen to requests from other processes (similar to TCP ports)

What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level. a. Synchronous and asynchronous communication b. Automatic and explicit buffering c. Send by copy and send by reference d. Fixed-sized and variable-sized messages

- Synchronous – allows for a rendezvous between sender and receiver, easier on prog. Level
- Asynchronous – messages may be received when they are not relevant anymore, more difficult on prog. Level as well as system level (requires kernel buffering)
- Automatic buffering – ensures that entire message is received, leads to synchronous communication
- Explicit buffering – buffer length is set so recipient might have to wait until buffer is freed
- Send and copy – better for network generalization and synchronization issues
- Fixed size effort – easier to implement on kernel level but require more effort by programmer
- Variable size effort – more complex on kernel level, easier for programmer

THREADS

States the advantages and disadvantages of shared communication

Can a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single process system Briefly explain your answer

- Multiple kernel threads provide better performance in case a single user thread has an error – one to many threading ensures the program keeps running
- Context switching between kernel threads has almost the same overhead as process context switching

Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single processor system?

No, the OS sees only a single process and will not schedule the different threads of the process on separate processors. No performance benefit

50% of a programs execution time is improved by a factor of 4. The remaining 50% is improved by a factor of 2. Calculate the speedup of the program

Let x be the execution time before improvements.

x/2 is improved by a factor of 4 resulting in x/8 execution time

The rest x/2 is improved by a factor of 2 resulting in x/4 execution time

Speedup= $\frac{x}{\frac{x}{8} + \frac{x}{4}} = \frac{8}{3} \approx 2.67$

In a multi-threaded server, why is threadpool better than one thread per request?

Where are mutex locks used in the threadpool strategy

Where are the condition variables used in the threadpool strategy

Provide three programming examples in which multithreading provides better performance than a single threaded application

- Web server environment – different threads perform different tasks for different users
- Graphical user interface

Provide three examples in which multithreading provides no benefit over a single threaded application

- *Linear programming – e.g. computing the factorial, sorting numbers, etc.*
- *Shell programs*

What are the differences between user-level threads and kernel-level threads, and under what circumstances would you use one over the other?

- *User level threads*
 - *Existence is unknown to kernel, as such they are less overhead to create, scheduled by thread library*
 - *Better if kernel time is shared, i.e. frequent context switching between user threads is less overhead than switching between kernel threads*
- *Kernel level threads*
 - *Existence is known to kernel, more overhead to create, scheduled by kernel*
 - *Better for multiprocessor environment because of parallelism*

Describe the actions taken by a kernel to context-switch between kernel level threads

What resources are used when a thread is created? How do they differ from the resources needed when a process is created?

- *Code section, data section and other OS sections such as open files are shared resources between threads*
- *Processes have their own code, data and OS sections*

Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Which of the following components of program state are shared across threads in a multithreaded process? a. Register values b. Heap memory c. Global variables d. Stack memory

- *Heap memory and global variables*

Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

Is it possible to have concurrency but not parallelism? Explain.

Using Amdahl's Law, calculate the speedup gain for the following applications: • 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores • 67 percent parallel with (a) two processing cores and (b) four processing cores • 90 percent parallel with (a) four processing cores and (b) eight processing cores

R1)

transport layer packet = segment
link layer packet = frame
network layer packet = datagram

routers and link-layer switches are called packet switches. The difference is, their forwarding decision is based on different criteria. Routers forward based on the values in the headers of the datagram, whereas link-layer switches base their forwarding decision on the values in the link-layer frame. This is why they are called LINK-LAYER SWITCHES

R3)

Forwarding refers to router-local action of transferring a packet from an input link interface to the appropriate output link interface. Forward takes place at timescales of nanoseconds, in hardware

Routing refers to the network-wide process that determines the end to end paths that packets take from source to destination

R4)

The forwarding table within a router maps destination addresses to that particular routers outbound links. The table uses longest prefix matching rather than trying to match the entire address.

Once the most appropriate outbound link has been found, the packet is directed to that outbound link.

P7)

interface 0 - 10000000-11111111
interface 1 - 10000000-10111111
interface 2 - 11100000-11111111
interface 3 - 00000000-01111111

P8)

223.1.17/24

Subnet 1 supports at least 60 interfaces - 223.1.17.2/26
Subnet 2 supports at least 90 interfaces - 223.1.17.1/26
Subnet 3 supports at least 12 interfaces - 223.1.17.??/26

Subnet 1 (60) - 11011111.00000001.0001001.01000000/26
Subnet 2 (90) - 11011111.00000001.0001001.10000000/25 ($2^7 - 2$) 126
Subnet 3 (12) - 11011111.00000001.0001001.00000000/26 ($2^6 - 2$) 62

128 64 32 16 8 4 2 1
00111111

Confused about the binary/denary translation here

The mask /n is the first n bits, but the bits to the left are the most significant, rather than least

P11)

128.119.40.128/26
128.119.40.129

Mask

11111111.11111111.11111111.00000000

IP

10000000.01110111.00101000.01000000

10000000.01110111.00101000.01|00|0000 = 128.119.40.64/28

10000000.01110111.00101000.01|01|0000 = 128.119.40.80/28

10000000.01110111.00101000.01|10|0000 = 128.119.40.96/28

10000000.01110111.00101000.01|11|0000 = 128.119.40.112/28

P17)

IP layer stamps ID number on packets sequentially

First IP packet generated is random number, after that, sequentially

a - we will note down the ID numbers of packets that we encounter, by keeping a hash P

- When we encounter an ID, check P[id-1] to see if we have seen it before.
- If not, store it in P[id].
- If we have seen this stream before, delete P[id-1] and add P[id]

After running this algorithm for a period of time, the number of keys in the hash will indicate how many packet streams have been active during the time period

- b) No. We would not know which stream the packet belonged to, so we would not be able to determine if an encountered packet was a new host communicating with the outside world, or a packet from an existing stream

Operating Systems

18 October 2020 15:17

Objectives:

- What is an OS?
- Main functions
- How is an OS designed?

An OS is a software that acts as an intermediary between a user and the hardware

The design varies on the purpose – e.g. PC OS are designed to be user friendly

The main job of an OS is to allocate hardware resources to user processes, and control their execution.

Goal: avoid failures and errors, e.g. two programs attempting to write at the same memory location simultaneously

The most basic function of an operating system is to load a program into memory, executing the program, and then stopping the program

While executing, the program may have to do I/O operations – for efficiency and protection, users cannot control devices directly

Programs may require read/write access to directories. OS provides system calls, which can be used to achieve this.

OS also allows communications between various processes. Inter-process communication happens through shared memory and another method that we'll discuss in Processes.

The OS also handles errors. E.g. power failure.

The OS handles accounting, by checking how many processes are running and what resources they're using. This info is gathered by process control blocks.

I/O

Overview of an I/O operation

- To start an I/O operation, the device driver loads the appropriate registers in the **device controller**, which in turn examines the contents of the registers to determine what action to take.
- The controller then starts the transfer of data from the device to its local buffer.
- Once the transfer is complete, the device controller informs the device driver that it has finished its operation.
- The device driver then gives control to other parts of the OS.

The controller informs the device driver that it has finished its operation via an **interrupt**.

Interrupts

Overview of an interrupt

- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the **system bus**.
- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

- The fixed location usually contains the starting address where the service routine for the interrupt is located.
- The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Every OS has its own interrupt mechanism, but some features are common:

The interrupt must transfer control to the appropriate interrupt service routine. This is done through a table of pointers to interrupt routines. This is stored in low memory (first few locations). These locations hold the addresses of the interrupt service routines for the various devices. This is called an **interrupt vector**.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt.

If the interrupt routines needs to modify the processor state, it must explicitly save the current state and then restore that state before returning.

After the interrupt is serviced, the save return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Storage

The CPU can only load instructions from memory, so any programs must first be loaded into memory to be run.

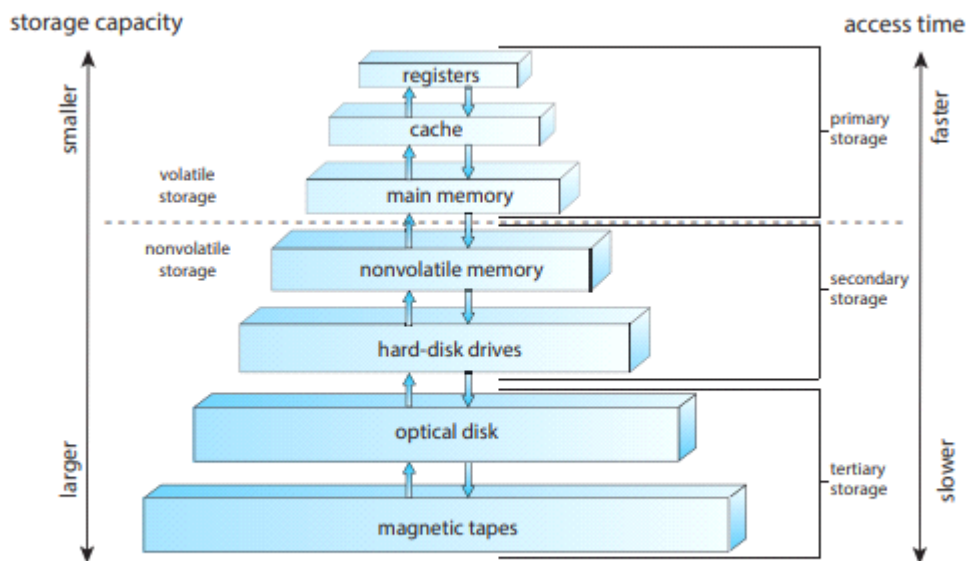


Figure 1.6 Storage-device hierarchy.

Kernel

- Kernel is the core of an OS
- It is loaded into the main memory at system start-up
- It is the process running at all time on the computers
- There are certain function which only a kernel can perform – e.g. memory management, process scheduling, file handling.

The part of memory where the kernel runs is called the kernel space. The divide between kernel and

user space is strict to protect the kernel from user operations.

System Calls – this is how user programs request the kernel to make certain privileged calls. System calls are low level operations.

To distinguish between user and kernel ops we use the mode bit, 0 or 1 depending on whether we are running in user or kernel mode.

Since the OS and its users share the hardware and software resources, an OS must ensure that an incorrect program cannot cause other programs to execute incorrectly.

Therefore, we distinguish between the execution of OS code and user code.

user mode

kernel mode (supervisor mode, system mode, privileged mode)

This is done with the mode bit.

When we're in user mode executing user code, and we need a service from the OS via a system call, we need to transition from user mode to kernel mode.

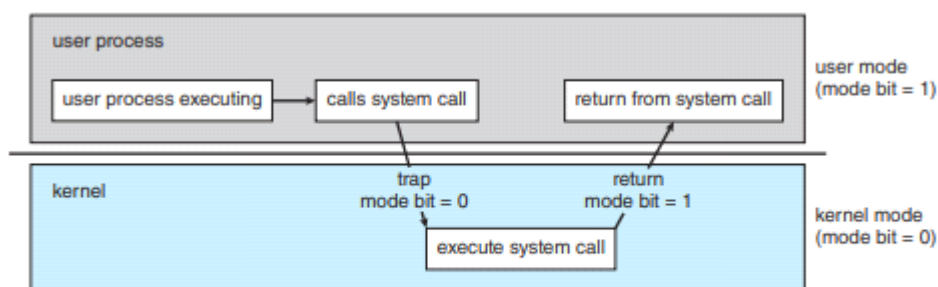


Figure 1.13 Transition from user to kernel mode.

Operating System Structure

Monolithic Structure

The simplest structure for organising an OS is no structure at all.

All functionality of the kernel is in a single, static binary file that runs in a single address space.

Advantages

Simple

Very little overhead in system-call interface

Communication within the kernel is fast

Disadvantages

Difficult to implement and extend

Can't be modified during run time

Layered Design

- Can be achieved through separation of different layers.
- The bottom layer is the hardware and the top layer is the user.
- Layer K uses the service of layer K+1 and provides to layer K+1

Advantages

- can modify the structure of one layer without modifying any other layers.

- It is simpler to construct a layered structure than a monolithic one
- Easier to debug
- Clear interfaces between layers

Disadvantages

- Defining layers is difficult
- System calls access multiple layers to execute, which adds overhead. So system calls can be slower on layered structures
- Layers below can never require the services of a layer above

Microkernels

The Mac operating system modularised the kernel. Removed all non-essential components from the kernel and implemented them as either system or user-level programs.

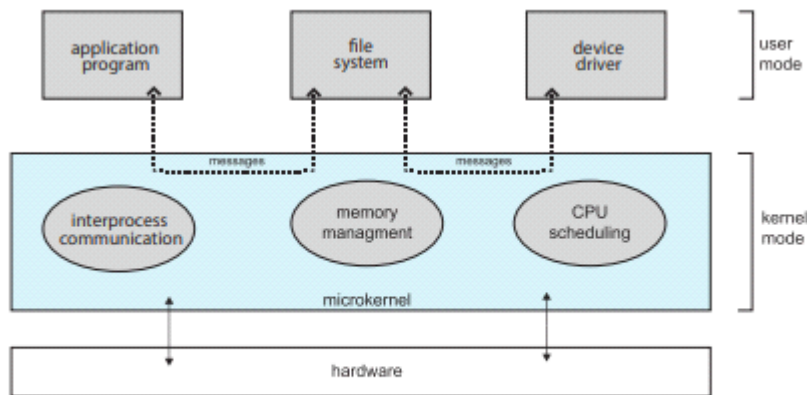


Figure 2.15 Architecture of a typical microkernel.

Advantages

- This results in a smaller kernel
- Extending the OS is easy, services are running in the user space so more secure and reliable.

Disadvantages

- Performances hampered due to increased system call overhead.

Modules

Most modern OS's use Loadable kernel modules (LKM's) – similar to the layered approach but differs because any module can call any other module – similar to microkernels because the kernel only provides core functionality, while other services are implemented dynamically as the kernel is running.

The kernel has a set of core components that can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX.

The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.

Linking services dynamically is preferable to adding new features directly to the kernel which would require recompiling the kernel every time a change is made.

Thus for example we might build CPU scheduling and memory management algorithms directly in to the kernel and then add support for different file systems by way of loadable modules.

Result resembles a layered system in that each kernel section has defined, protected interfaces, but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules, but it is more efficient because modules do not need to invoke message passing in order to communicate

Advantages

- More flexible than a layered system because any module can call any other module
- More efficient than the microkernel approach because modules do not need to use message passing in order to communicate
- Do not need to recompile the kernel every time a change is made since services are linked dynamically

Hybrid Systems

Very few systems adopt a single strictly defined structure. E.g. linux is monolithic in that the operating exists in a single address space, providing efficient performance, however it is also modular in that new functionality can be dynamically added to the kernel.

Introduction to C

18 October 2020 15:12

Objectives:

1. Functions
2. Pointers
3. Arrays
4. Structures
5. Dynamic memory allocation

Pre-processing directives start with #:

- `#include <filename>` or `#include "filename"` includes the content of a filename
- `#define name replacement`
- Will replace occurrences of name with replacement

```
Int main() {  
    Printf("hello");  
}
```

`%d` for integers, `%f` for floats, `%s` for chars

To execute – **`gcc filename.c -o executablename`**

`./executablename`

Declare variables prior to use like

```
int I;
```

To take user input:

- `scanf() t`
- `scanf("%d",&I);`

For loop syntax is standard

```
for (I=0, I<10, I++) {}
```

Anything non zero is true in C

`%` is remainder

Always define the function prior to calling it, or define a prototype before and call it after.

Pointers, Addresses, and Dereferencing

```
Int x=30;
```

`&x` # would print the hex value of the location

`x` # would print 30

`*(&x)` # would dereference the location of the variable and print 30

```
Int *ptr; # declares a pointer
```

i.e. it is going to store the address of an integer

If we want to store the address we can do

```
ptr=&x;
```

So instead of de-referencing the address, we can print *ptr to print 30.

Function(&x) # is a call by reference – it passes the location of x

In function we can have float function(*x) to show that we're receiving a pointer

Then we can do *x=*x+2 to change the value in the location referenced by the pointer. This will now be changed in the calling function.

Arrays

Int x[10] # declares an array

X[l] = 4 # stores 4 at the l'th location

Array elements are stored in contiguous memory locations

E.g.

&x[0] will be next to &x[1] will only be different by the number of bytes used to store an integer – sizeof(int)

&x[0] or x will give the base address

(x+I) # is the same as (&x[I])

*(x+I)=2; # same as x[I] = 2

Structures

Group together variables of different types in a single block

```
Struct employee {  
    Char *name;  
    Int age;  
}
```

```
Struct employee emp1  
Emp1.name = "john";  
Emp1.age = 50;
```

If you're using a pointer to a struct, then use

ptr->age # to get variables, or alternatively, dereference and call normally e.g.

```
struct employee emp1 = *ptr;
```

```
Int age = emp1.age;
```

Dynamic Memory Allocation

If memory needs to be assigned at runtime, dynamic memory allocation is to be used.

```
Int *ptr;  
Ptr = (int *) malloc(20*sizeof(int));
```

Memory can be allocated dynamically using malloc, calloc, realloc functions

Malloc returns a void pointer pointing to the base address of the block of memory

Calloc has 2 args – number of units and size of each unit

```
Ptr = (int *) calloc(20, sizeof(int));
```

If we want to grow an existing array, then we can use realloc to keep the existing data in the memory block

Check for null pointer to see if we have successfully allocated a block of memory

Use free to free up memory

Processes

18 October 2020 16:48

Objectives:

- Process definition
- Process in memory
- Program control block
- States of a process
- Process scheduling

A process is a program in execution.

Programs sit on the disk as an executable, and are passive – they just contain data and instructions

Processes are active. They are loaded into memory.

There are multiple processes that are active in a system at any one moment - system level processes that execute system code, and user processes that execute user code. All these processes can potentially execute concurrently by multiplexing on a single CPU

The OS is responsible for these activities in process management

- Scheduling processes and threads on the CPU
- Creating and deleting both user and system processes
- Suspending and resuming process
- providing mechanisms for process synchronisation
- Providing mechanisms for process communications

One program can create multiple processes.

In a multithreaded program, there are multiple program counters that each point to the next instruction of a given thread

A Process in Memory

There is a virtual address space of a process that is allocated, in which the following reside:

- Text – stores the instructions; the executable code
- Data – stores the global variables
- Heap – dynamically allocated memory during program run time
- Stack – stores local variables and function parameters such as return address of a function

The space between the stack and heap allow them to grow and shrink during the program run-time.

Process State Changes

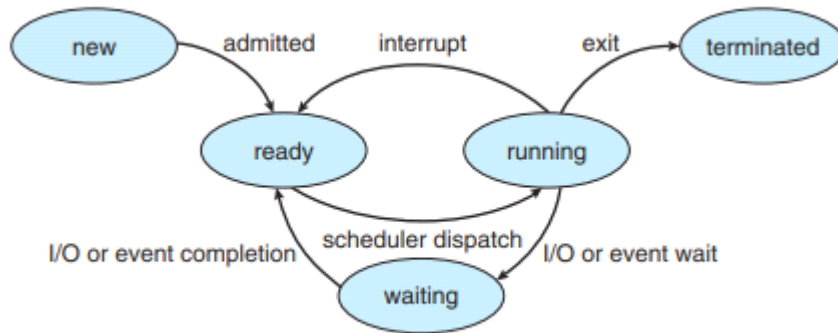


Figure 3.2 Diagram of process state.

- New - the process is being created
- Running - instructions are being executed
- Waiting - the process is waiting for some event to occur (such as an I/O completion)
- Ready - the process is waiting to be assigned to a processor
- Terminated - the process has finished execution

Process Control Block

Each process is represented in the OS by a PCB. It contains information associated with a specific process:

- Process state – running waiting ready
- Program counter – location of the next instruction to be executed for this process
- CPU registers – contents of the CPU registers. The registers can vary in number and type, depending on the architecture. They include accumulators, index registers, stack pointers, and general purpose registers. State information must be saved when an interrupt occurs so that the system can resume its current state after the interrupt has been serviced.
- CPU scheduling information – process priority, pointers to scheduling queues.
- Memory management information – memory allocated to the process
- Accounting information – CPU used, time since state
- I/O status – list of open file / I/O devices

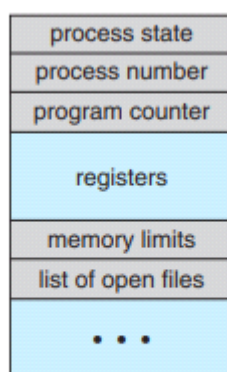


Figure 3.3 Process control block (PCB).

Process Creation

- A process may be created by another process
- Children processes can in turn create other processes forming a process tree
- Process are usually identified and managed via a PID

All user processes are rooted at PID 1 – the init process

The parent can choose various option

resource sharing options

- Files are shared between parents and children
- A subset of files are shared
- No resources are shared

execution options

- The parent may wait for the child to finish, or can continue concurrently

address space options

- The child's address space is a duplicate of that of its parents (has the same code, data and stack as parent)
- The child loads a new program into its address space

In unix, the fork() system call creates a new process with duplicate address space of the program

Process Termination

- Terminates automatically with exit or when last line is called
- Returns a status value to parent
- All resources of the process are released by OS

After a child terminates, and before the parent receives the code, the child is a zombie process. During the phase, all resources are released

What happens if a parent exists without invoking wait? The child is now an orphan process

In UNIX systems, the init process is assigned the parent

The init process periodically issues the wait() to collect the exit status of all orphan processes. This allows the exit status to be collected and released the orphans PID and process table entry

A parent process may terminate the execution of child processes using the abort() system call. This may happen if:

- If the child has exceed its allocated resources
- Task assigned to child is no longer required
- The parent is exiting and the OS does not allow a child to continue if its parent terminates – cascading termination

Process Scheduling

- To maximise CPU use, quickly switch process onto CPU for time sharing
- Process Scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes – job queue, ready queue, device queue

Short term scheduler

takes jobs from the ready queue, invoked once every 100ms. Schedules jobs from that queue

Long term scheduler

- When a process is finished executing and leaves the system. Maintains CPU bound and I/O jobs in main memory. Occurs much less frequently, controls the degree of multiprogramming, i.e. number of processes in memory. Tries to maintain a balance of jobs so that the system is stable and the queues don't grow out of hand. We have to strike a balance between the 2 types of jobs.

- Linux and windows don't have a long-term scheduler as they just dump all processes on the short-term scheduler.
- Linux/Windows assumes that the user themselves will notice a degradation in performance and kill jobs. Therefore, you are acting as the long term scheduler.

Context Switching

Interrupts cause the OS to change a CPU core from its current task and to run a kernel routine.

Such operations happen frequently. The system needs to save the current context of the process running on the CPU core so that it can restore that context when it is done processing the interrupt. The context is represented in the PCB of the process.

Switching context requires doing a state save, and then a state restore of a different process.

The kernel saves the context of the old process in the PCB, and loads the saved context of the new process scheduled to run.

Context switch time is pure overhead, because the system does no useful work while switching.

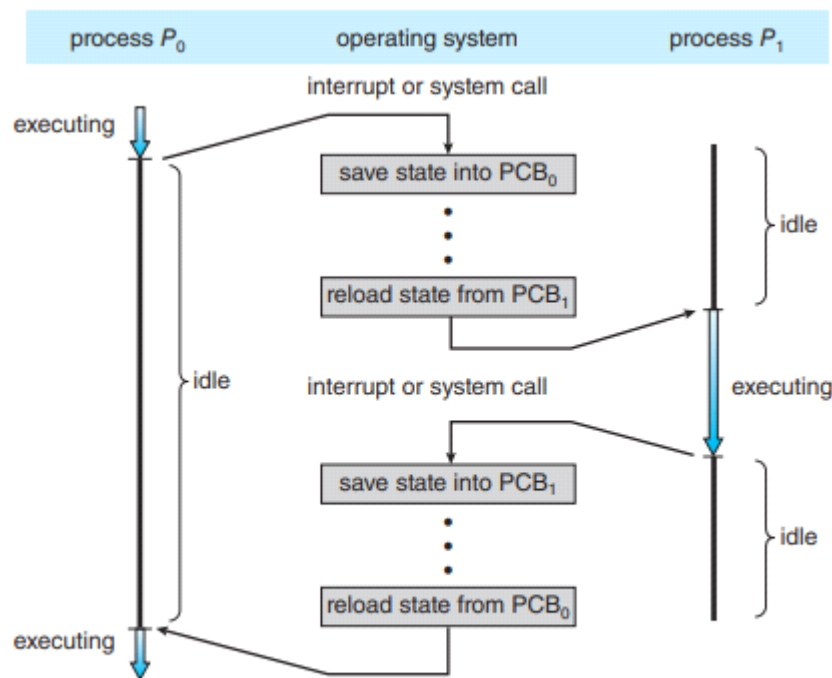


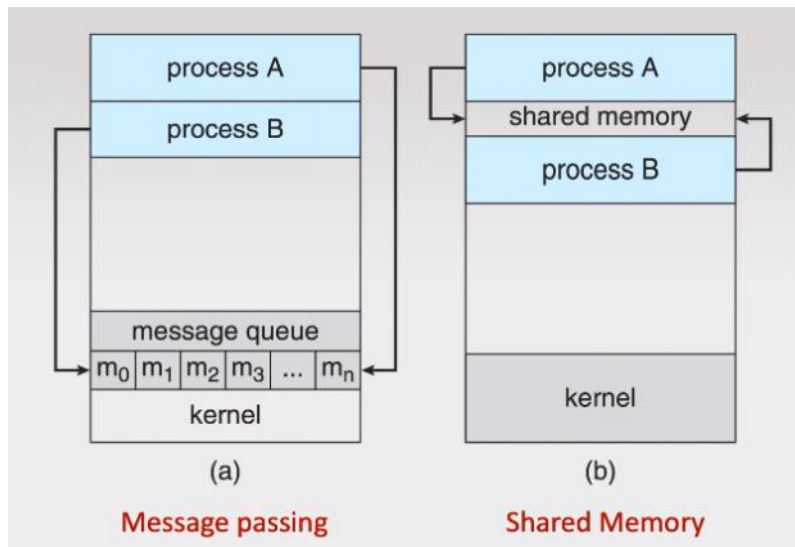
Figure 3.6 Diagram showing context switch from process to process.

Inter-Process Communication

Provide mechanisms for processes to communicate with each other, e.g. they're sharing data to complete a task

Two ways of inter-process communications

1. Shared memory
2. Message passing



Message Passing

Repeated system calls cause overhead

Useful for passing smaller amounts of data as no conflicts need to be avoided

More easily implemented in distributed systems.

Shared Memory

Less overhead because the only system calls necessary are the initial calls to allocate the shared memory

Can be faster due to the lack of need for kernel intervention in system calls.

Both methods are commonly used

Shared Memory

One of the processes has to create a shared space, with the help of the kernel.

The programmer needs to make sure that both processes don't try to write to the shared memory at the same time.

A process is either a producer or a consumer at any given moment.

1. Consumer should wait when buffer empty or it will read garbage.
2. Producer must wait when the buffer is full

We can use a circular queue to solve the above two problems.

Why do we use a circular queue to solve this problem?

A pseudo producer

```
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;
}
```

A pseudo consumer

```
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 */
}
```

// Note: the buffer space is not fully utilized in this method

Message Passing

1. Needs a send and a receive operation
2. These are implemented using system calls
3. The kernel has to make sure the messages are sent and received correctly

Link implementation – direct or indirect

Synchronisation between send and receive

Buffer size representing the communication link

Direct Communication

1. Processes must name each other explicitly
2. Properties of communication link
3. Links are established automatically
4. A link is associated with exactly one pair of communicating processes
5. Can't hardcode the PID because every time the process runs its id can change

Indirect Communication – preferred

Sent and received through a mailbox, using a mailbox id. The receiver then reads from the box

Similar to shared memory system but the mailbox is managed by the kernel

Synchronous or Asynchronous

Blocking is considered synchronous

- Blocking send – the sender is blocked until the message is received
- Blocking receive – the receiver is blocked until a message is available

Non-blocking is considered asynchronous

- Non-blocking send – the sender sends the message and continues
- Non-blocking receive – the receiver receives a valid message or a null message

Communication link is a buffer. The implementation of send and receive depends on the capacity of this buffer.

- Zero capacity – the queue has a maximum length of zero, so sender must block until the recipient receives the message
- Bounded capacity – the queue has a finite length, when full the sender must block
- Unbounded capacity – the queue length is potentially infinite – sender never blocks

Communicating back to the parent with shared memory - in lecture at [20 mins](#)

Ordinary Pipes

- Allow simple one way communication
- Connects the output of one process to the input of another

Named Pipes

- A problem with ordinary pipes is that they die when the process terminates
- Named pipes are more powerful – no parent-child relationship is required
- And once established, they persist

Threads

06 November 2020 12:17

What are Threads?

A thread is unit of CPU execution

We can make a process multi-threaded. Each thread can handle a separate task

Threads share:

- global variables
- open files
- data
- heap
- signals

Threads do not share:

- Stack
- Registers

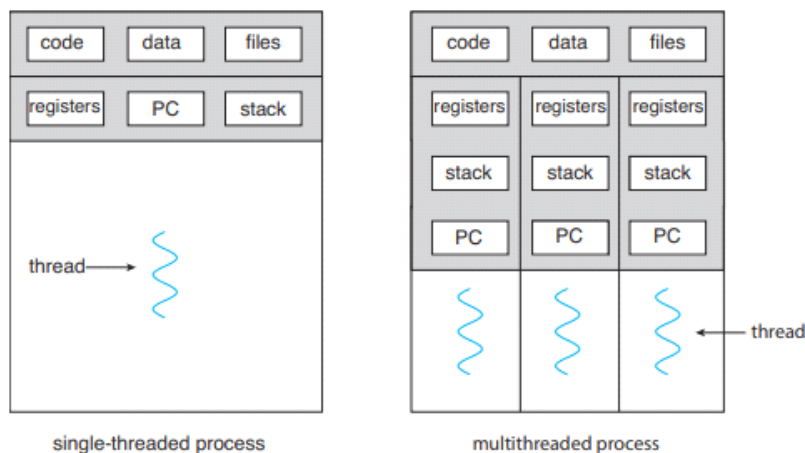
Each thread needs a *thread id*, a *program counter*, a *register set* and a *stack*

- Thread creation has less overhead
- Process creation has more overhead

Modern web-servers use multi-threading due to high traffic

Benefits of Threads

- Economy - cheaper than process creation because of lower overhead. Thread switching is faster than context switching
- Scalability – large number of concurrent tasks
- Responsiveness – allows threads to continue executing when others are blocked
- Resource sharing – resources are shared so easier on memory usage



Concurrency vs Parallelism

Concurrency

Supports more than one task making progress – a single CPU may appear to be running tasks

concurrently by interleaving their execution

Parallelism

Different processes running simultaneously on multiple cores

1. Data parallelism distributes subset of the same data across multiple cores performing the same operation on each core – summing the contents of an array
2. Task parallelism splits threads performing different tasks across multiple cores

Thread Synchronisation

1. When multiple threads write to the same location, we must synchronise the threads
2. Synchronisation ensures that one thread does not overwrite the contents written by the other thread
3. Otherwise we end up with a race condition
4. To synchronise we can use mutex locks – mutual exclusion locks
5. Each thread must first acquire a lock to do an update

Amdahl's Law

A formula that identifies potential performance gains from adding additional computing cores to an application that has both series and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follow.

The diagram shows the Amdahl's Law formula: $speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$. The formula is enclosed in a light blue box. Red circles highlight the '1' in the numerator, the ' S ' in the denominator, and the fraction ' $\frac{(1-S)}{N}$ '. Red arrows point from these circles to text labels: the top arrow points to 'Time taken before parallelising', the bottom-left arrow points to 'Time taken to run the serial part. $0 \leq S \leq 1$ ', and the bottom-right arrow points to 'Time taken to run the parallelisable part with N cores'.

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

Time taken before parallelising

Time taken to run the serial part.
 $0 \leq S \leq 1$

Time taken to run the parallelisable part with N cores

For example, assume we have an application that is 75% parallel and 25% serial. If we run this application on a system with two processing cores, we can get a speedup of 1.6 times. If we add an additional two cores, making a total of 4, the speedup is 2.28 times.

One interesting fact about Amdahl's law is that as N approaches infinity, the speedup converges to $1/S$. E.g. if 50% of the application is performed serially, the maximum speedup is 2 times, regardless of the number of cores we add.

This is the fundamental principle behind Amdahl's Law: the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.

Thread Types

User-Level Threads

The user-level threads are implemented by users and the kernel is not aware of the existence of

these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter(PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronization for user-level threads.

Advantages of User-Level Threads

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Disadvantages of User-Level Threads

- Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
- The entire process is blocked if one user-level thread performs blocking operation.

Kernel-Level Threads

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Advantages of Kernel-Level Threads

- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel routines can also be multithreaded.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Disadvantages of Kernel-Level Threads

- A mode switch to kernel mode is required to transfer control from one thread to another in a process.
- Kernel-level threads are slower to create as well as manage as compared to user-level threads.

Thread Models

Many-To-Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

Many-to-One Model

Many user level threads mapped to a single kernel thread

- Advantage – less overhead
- Disadvantage – multiple threads may not run in parallel because only one may be in kernel at a time

One blocking thread causes all to block

Multiple user level threads are mapped to a smaller number of kernel level threads
User threads take turns to use the kernel

Programmers can decide how many kernel threads to use and then can run in parallel

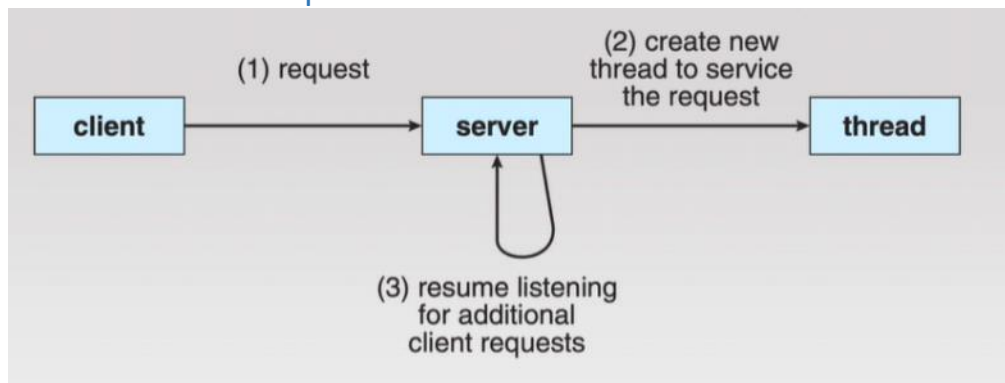
One-to-One Model

Each user level thread maps to a kernel model

- Advantage – other threads can run even when one thread is blocking
- Different threads can be run on different cores in parallel
- Disadvantage – a user process can create a large number of kernel threads slowing the system down

Multi-Threading Strategies

One Thread Per Request



Server creating a separate thread to handle each client request

Problems

Making a new thread per client request can take a long time if kernel level threads
Large number of concurrent threads, slowing down the system

Thread Pool

- Main thread creates worker threads
- Each incoming request is added to a queue
- The workers pull requests from the pool queue

Advantages of Thread Pooling

- Usually slightly faster to service a request with an existing thread rather than creating a new thread
- Allows the number of threads in the applications to be bound to the size of the pool. This size is decided by the programmer taking into account the number of processing cores, memory in use, expected number of concurrent client requests

When there are no jobs for the thread, it continues to try processing jobs. This is inefficient.
One way of solving this problem is to use condition variables...

Condition Variables

- We can solve this using condition variables
- A thread can use `wait()` to wait on a condition variable
- Until some other thread signals the variable using `signal()` or `broadcast()`

Signal Handling

Signals are used in UNIX systems to notify a process about the occurrence of certain events

1. Synchronous – internally generated by a process e.g. division by 0 or illegal memory access
 2. Asynchronous – externally generated e.g. terminating a process using ctrl+c sends the signal SIGINT
- All signals follow the same pattern – a signal is generated by an event, the signal is delivered to the process to which it applies and once delivered the signal is handled
 - The signal can be handle in two ways, the kernel runs a default kernel handling routine
 - By user-defined signal handling we generally override a general function used by the kernel

Synchronisation

03 April 2021 13:43

Objectives:

- Critical section problem
- Solution using different techniques
- Classic synchronisation problems

We introduced synchronisation using mutex locks while discussing threads

Why Synchronise?

Concurrently running threads/processes can cause a race condition when trying to update shared variables

Only one of the values is preserved - it is a race to see which one is last

We can avoid this using proper synchronisation

Terminology

- process - threads and processes
- critical section - the part of the code where the shared variables are updated by the process
- mutual exclusion - when one process is in a critical section, no other process should be allowed to be in its critical section

Critical Section Problem

- Consider a system with n processes.
- Each process has a critical section that updates shared variables.
- When one process is in its critical section, we want that no other process can be in its critical section
- The problem is to design a protocol that the processes can use to synchronise their activity so that they can share data
- Each process must request permission to access its critical section
- The code implementing this request is the entry section
- The critical section may be followed by an exit section
- The remaining code is the remainder section

A solution must satisfy 3 constraints

- **Mutual exclusion** - no 2 processes can access their critical sections simultaneously
- **Progress** - if no process is executing its critical section and then some processes wish to enter their critical sections, then only those processes that are not executing their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
- **Bounded waiting** - there exists a bound 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

Race Condition Example

- Consider as an example a kernel data structure that maintains a list of all open files in the system.
- This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list).
- If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Peterson's Solution

Restricted to two processes but can be extended to any number

Consider p0 and p1

There are two shared items:

1. int turn;
2. boolean flag[2]

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

    /* critical section */

    flag[i] = false;

    /*remainder section */
}
```

Figure 6.3 The structure of process P_i in Peterson's solution.

1. The variable turn indicates whose turn it is to enter the critical section
2. If $turn == i$, then process P_i is allowed to execute in its critical section .
3. The flag array is used to indicate if a process is ready to enter its critical section.
4. For example if $flag[i]$ is true, P_i is ready to enter its critical section.
5. If $flag[i]$ is true, P_i is ready to enter its critical section.

To enter the critical section, process P_i first sets $flag[i]$ to be true, and then sets turn to the value j , thereby asserting that if the other process wishes to enter the critical section, then it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

To prove that this solution holds, we need to prove:

- 1) mutual exclusion is preserved
- 2) The progress requirement is satisfied
- 3) The bounded-waiting requirement is met

See page 263 of OS textbook (PDF page 338)

Problems with Peterson's Algorithm

Employs Busy Wait

May fail in modern architectures because independent read and write operations may be reordered

Hardware Based Solutions

A more practical technique to solve the CS problem involves using the idea of locking.

Two processes cannot have a lock simultaneously.

Locking should be performed atomically - meaning the lock is single unit and thus non-interruptible

test_and_set Instruction

Returns the original value of passed parameter

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Figure 6.5 The definition of the atomic test_and_set() instruction.

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

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

Figure 6.6 Mutual-exclusion implementation with test_and_set().

This is atomic, so if two test_and_set() instructions are executed simultaneously on different cores, then they will be executed sequentially in some arbitrary order. If the machine supports test_and_set() then we can implement mutual exclusion by setting a boolean lock.

compare_and_swap Instruction

Operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

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

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Figure 6.7 The definition of the atomic compare_and_swap() instruction.

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

Figure 6.8 Mutual exclusion with the `compare_and_swap()` instruction.

The CAS instruction operates on three operands. The operate value is set to `new_value` only if the expression `(*value == expected)` evaluates to true. Regardless, CAS always returns the original value of the variable. It is always executed atomically. Thus if two CAS instructions are executed simultaneously on different cores, they will be executed in some arbitrary sequential order.

Software Locking

Mutex Locks

A mutex lock has a boolean variable `available` whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Lock Contention

Locks are either contended or uncontended. A lock is considered contended if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered uncontended. Contended locks can experience either high contention (a relatively large number of threads attempting to acquire the lock) or low contention (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications

Short Duration

Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration. But what exactly constitutes a short duration? Given that waiting on a lock requires two context switches— a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available— the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

Disadvantages of Mutex Locks

- Use busy waiting; any process trying to acquire the lock must loop indefinitely, until it can acquire the lock. This wastes CPU cycles that could be used by other processes.

Advantages of Mutex Locks

- No context switch is required when we acquire the lock. Context switching can have considerable overhead

Semaphores

Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.

Mutex locks are the simplest synchronisation tool. A semaphore behaves similarly to a mutex lock but is more robust.

A semaphore S is an integer variable that is accessed only through two standard atomic operations:

`wait()`

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

`signal()`

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

All modifications to the semaphore have to be atomic, so that no two processes can simultaneously modify the same semaphore value. In addition, the testing of S in `wait` as well as its possible modification must happen without interruption.

Synchronisation Issues

Deadlock

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, we have a deadlock

Starvation

- Occurs when a specific process has to wait indefinitely while others make progress - this is the opposite of bounded waiting
- Starvation occurs when multiple processes are waiting on a semaphore and the signal call wakes up the same process again and again
- To avoid such starvation, the signal should randomly pick the process to wake up

Priority Inversion

When a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process, or chain of processes.

Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is pre-empted in favour of another process with a higher priority.

This is known as priority inversion, and it can occur in systems that have more than two priorities.

We can avoid priority inversion by implementing a priority inheritance protocol.

According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

When they are finished, priorities revert to their original values.

Classic Synchronisation Problems

Bounded Buffer Problem

- n buffers, each can hold an item
- Producer produces an item and writes to a buffer while the consumer consumes an item from a buffer
- Producer should not produce when all n buffers are full
- Consumer should not consume when all buffers are empty

Solution

Use three semaphores

```
/* shared semaphores */  
semaphore mutex=1; /* maintains exclusive access to the buffer */  
semaphore full=0; /* counts the number of full buffers */  
semaphore empty=n; /* counts the number of empty buffers */
```

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

Signal and wait are used to increment and decrement atomically

```
Consumer: 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

Allow multiple readers to read at the same time

Only one single writer can access the shared data at the same time

Readers are given preference over writers when no process is active

Writers may starve

This is the readers and writers problem, and we use it to test new synchronisation primitives

Solutions

```

/* shared variables */
semaphore rw_mutex=1; /* lock to allow at most one writer */
semaphore mutex=1;    /* lock to protect the read_count */
int read_count=0;     /* counts the number of reader processes */

```

The semaphores needed

```

writer: do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);

} while (true);

reader: do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);

```

Mutex prevents read count being updated incorrectly

If they are the first reader they unlock the rw_mutex to prevent any writers writing

Dining Philosophers Problem

- Philosophers spend their lives alternating between thinking and eating
- Consider a round table of 5 philosophers
- In the centre of the table is a bowl of rice
- There are 5 single chopsticks
- When a philosopher thinks, they do not interact with their colleagues.
- Occasionally a philosopher tries to pick up the chopstick to the left and right to eat
- A philosopher can only pick up one stick at a time
- When they get 2, they can eat without releasing the chopsticks
- When they finish, they can put down the chopsticks and eat]

Semaphore Solution

- Represent each chopstick with a semaphore
- A grab is equivalent to a wait() operation on that semaphore
- A release is equivalent to signal() operation

This could create a deadlock. Suppose all 5 philosophers are hungry at the same time and each grabs the left chopstick. All the elements of chopstick will equal to 0. When each philosopher tries to grab the right chopstick, they will be delayed forever.

Solutions to the Deadlock:

- Allow at most 4 philosophers to be sitting at the table simultaneously
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (by doing so in the critical section)
- Use an asymmetric solution - that is, an odd numbered philosopher picks up their left, then right chopstick, whereas even do right then left

Scheduling

03 April 2021 13:43

Objectives:

- CPU scheduling is the basis of multi-programmed operating systems.
- By switching the CPU among process, we can make the computer more productive.
- In modern operating systems, it is kernel level threads, not processes, that are being scheduled by the operating system

process scheduling - in this section will mean general scheduling concepts

thread scheduling - will refer to thread specific scheduling problems

Basic Concepts

In a system with a single CPU core, only one process may run at a time. The goal of scheduling is to ensure that some process is running at all times.

A process is usually executed until it must wait, e.g. for an I/O request. In multiprogramming, we try to use this idle time for productive work. Several processes are kept in memory at the same time, and when one sits idle, the OS takes the CPU away from that process and gives the CPU to another process.

CPU-I/O Burst Cycle

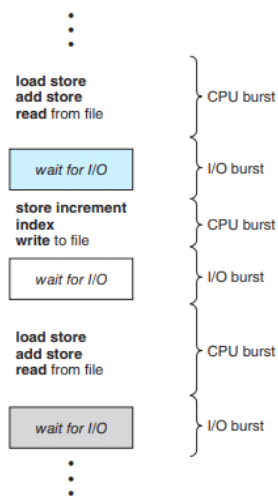


Figure 5.1 Alternating sequence of CPU and I/O bursts.

The success of CPU scheduling depends on an observed property of processes - process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.

Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

Eventually, the final CPU burst ends with a system request to terminate the execution.

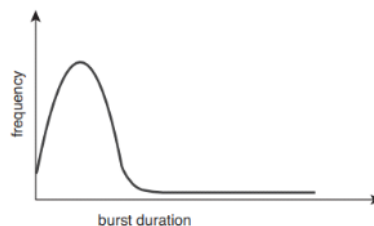


Figure 5.2 Histogram of CPU-burst durations.

CPU bursts vary greatly from process to process and across computers, but they tend to have a distribution similar to that of the graph in 5.2

A CPU-bound program might have a few long CPU bursts. The distribution can be important when implementing a CPU - scheduling algorithm.

CPU Scheduler

When the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.

The selection is carried out by the CPU scheduler, which selects a process from the processes in memory that are ready to execute, and allocates the CPU to that process.

The ready queue is not FIFO, since there are many scheduling algorithms - the queue can be a FIFO queue, but also an unordered linked list, a priority queue or a tree.

The records in the queues are generally PCB's of the processes.

Goals of CPU Scheduling

1. Maximise CPU utilisation
2. Maximise throughput
3. Minimise waiting time - amount of time a process is in the ready queue
4. Minimise turnaround time - amount of time to execute a particular process
5. Minimise response time - amount of time it takes from when a request was submitted until the first response is produced

Pre-emptive and Non-pre-emptive Scheduling

- **Non-pre-emptive** = CPU is given up voluntarily
- **Pre-emptive** = CPU is snatched

CPU scheduling happens in 4 situations:

- When a process switches from running to waiting
- When a process switches from running to ready
- When a process switches from waiting to ready
- When a process terminates

For 1 and 4, we have no choice in terms of scheduling . A new process must be selected for execution.

When scheduling takes place under 1 and 4, we say that the scheduling is Nonpreemptive or cooperative. Otherwise, it is pre-emptive (2 and 3).

- **Nonpreemptive scheduling** means that once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems use pre-emptive scheduling.
- **Pre-emptive scheduling** can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is pre-empted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

Scheduling Algorithms

First-Come-First-Served

Simplest algorithm. First process that requests the CPU is allocated the CPU. The implementation is based on a FIFO queue.

When the CPU is free, it is allocated the PCB at the head of the queue, and the running process is removed from the queue.

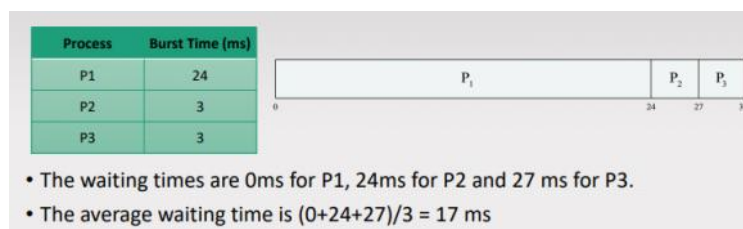
This algorithm is non-preemptive; once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by termination or requesting I/O.

This means that FCFS is troublesome for interactive systems.

Disadvantages

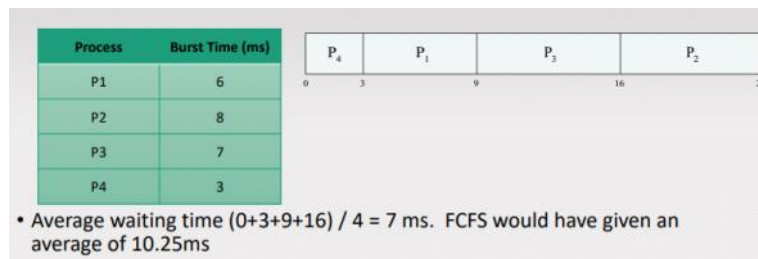
Average waiting time is quit long - may vary substantially depending on the CPU burst times

Can't be used for interactive systems where it is important that each process gets a share of the CPU at regular intervals



Shortest Job First

The process with the shortest next CPU burst is selected, if there is a tie then FCFS



Advantages

Provably optimal, in that it gives the minimum average waiting time - moving a short process before a long one decreases the waiting time of the shorter process more than the increased waiting time for the long process

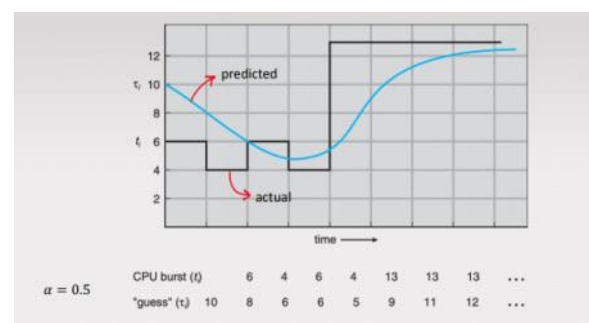
Disadvantages

We don't know how long the next CPU burst will be - we can only estimate

Exponential Moving Average for Predicting CPU Burst

- t_n = actual length of n^{th} CPU burst
- τ_{n+1} = predicted value of the next CPU burst
- $\alpha, 0 \leq \alpha \leq 1$
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

Expanding the recursion gives

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$


The SJF algorithm can be either pre-emptive or non-pre-emptive

- **Pre-emptive**: switch to newly arrived processes
- **Non-pre-emptive**: allow currently existing job to finish

Problems with Pre-Emptive Scheduling

- Can cause race conditions
- If a process is pre-empted while it is updating some shared data, this leaves the data in an inconsistent state
- Shared data should thus be updated within critical sections using proper synchronisation primitives

Priority Scheduling

SJF is a special case of priority scheduling in which a priority is associated with each process and the CPU is allocated the process with the highest priority in the queue.

In SJF, priority is predicted CPU burst time

Disadvantages

Starvation can occur, where very low priority processes may never get scheduled

Starvation can be solved by aging, in which we gradually increase the priority of processes that have been in the system for a long time

Picking Priority

We could determine priority by:

- Number of open files
- memory requirements
- ratio of average I/O burst to average CPU burst
- time limits

Round Robin

1. Each process gets a small unit of CPU time (time quantum q). After this time has elapsed, the process is pre-empted and added to the end of the ready queue
2. Generally the scheduler visits the processes in order of their arrivals
3. 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 $(N-1)*q$ time units for its next turn

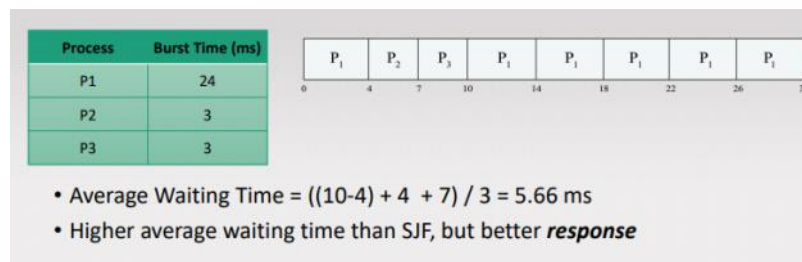
RR is pre-emptive. Timer interrupts every quantum to schedule next process

Performance

- q large \rightarrow similar to FCFS
- q small \rightarrow too many context switches cause a lot of overhead

q is usually 10-100ms, while context switches are usually $< 10\mu s$

If $q = 4ms$:



Deadlocks

02 January 2021 13:23

Objectives:

- Definition
- Conditions
- Deadlock detection
- Deadlock prevention and avoidance algorithms

What is a Deadlock?

A set of processes is in a deadlock when each process in the set is waiting for an event that can be caused only by another process in the set

The events we are interested in are acquisition or release of some type of resource

- Mutex locks
- CPU
- File
- I/O devices

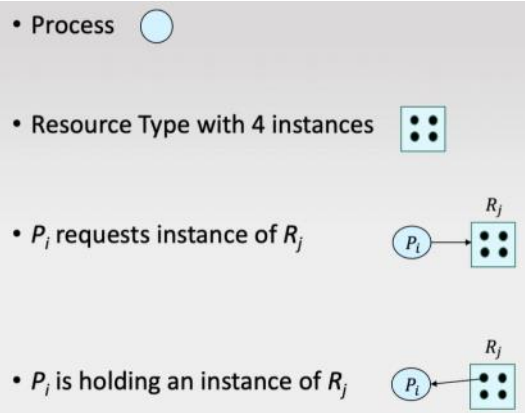
Necessary Conditions for Deadlock

- Mutual exclusion – only one process at a time can use a resource
- Hold and Wait – there must be a process holding some resources while waiting to acquire additional resources held by other processes
- No pre-emption – a resource can be released only voluntarily, and it cannot be snatched
- Circular wait – there must exist a subset of processes waiting for each other in a circular manner

Resource allocation graph

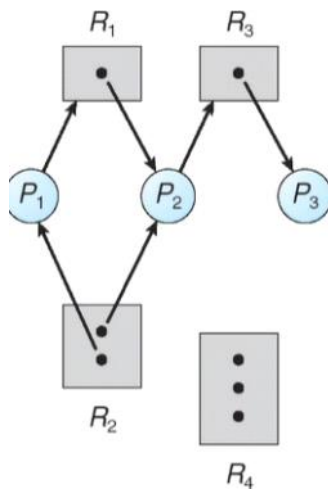
A directed graph $G=(V,E)$ where

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$



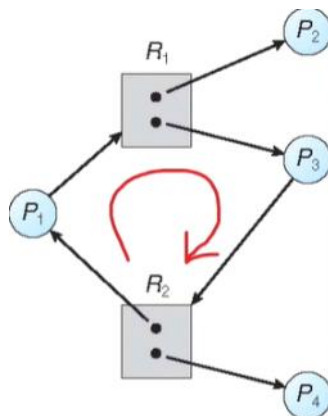
Resource Allocation Graph Example 1

- P1 has acquired an instance of R2 and is waiting for R1
- P2 has acquired an instance of R2 and is waiting for R3
- R3 has already been acquired by P3



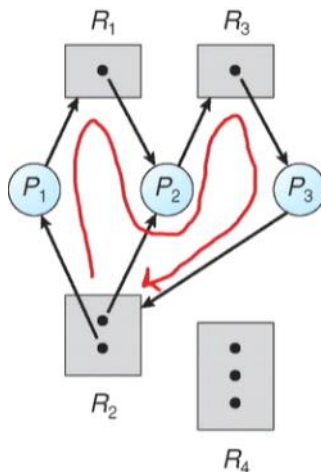
There is no deadlock in this graph. Once P3 finishes, it will release R3 and then finish, giving R3 to P2, which will then finish

Resource Allocation Graph Example 2



Resource Allocation Graph Example 3

There is a cycle in the graph, but there is no deadlock.



There is a cycle, and insufficient resources for any of the cyclic processes to finish, so they can't free up the resources needed by the other processes.

There is a deadlock

- If there is no cycle, there cannot be a deadlock
- If there is a cycle, there might be a deadlock

Algorithms for Deadlock Detection

	Allocation			Currently Available			Request		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P1	0	1	0	0	0	0	1	0	0
P2	1	1	0				0	0	1
P3	0	0	1				0	0	0

- Allocation represents current allocation of resource to Process
- Currently available represents how much is available in a system e.g. at the moment there are no resources available
- Requests represents pending requests - e.g. p2 is requesting 0 r1, 0 r2 and 1 r3 – this is pending

We also need a Boolean flag corresponding to whether a process has finished

Handling Deadlocks

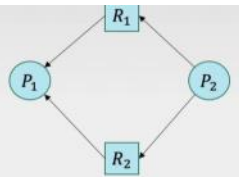
Deadlock Prevention

If we can ensure one of the necessary conditions doesn't hold, we can ensure the system is deadlock free

- Mutual exclusion – it is difficult to avoid this
- Hold and Wait – avoiding this means we have to ensure that when a process requests a resource, it does not hold any other resources
- No pre-emption - if we allow snatching of resources, we can avoid deadlocks
- Circular wait - we could give each resource a number and require that processes request resources in order of number

This is restrictive – harmless requests could be blocked e.g.

- Consider the following system:
 - P_1 and P_2 each requires resources R_1 and R_2
 - Resources are numbered: $n(R_2) > n(R_1)$
Each process must request resources in the increasing order
 - If R_2 is already held by P_1 , then the request of R_1 by P_1 will be blocked (not granted)
 - But it is safe to grant this request since there will be no cycle in the resulting system



Deadlock Avoidance

- Better than deadlock prevention
- Determine whether a request should be granted based on whether it will leave the system in a safe state
- Safe state is a state when a deadlock can't occur

How do we know if a state is safe?

We need to know what states can arrive in the future

Deadlock Avoidance Algorithm

- Needs advanced information on resource requirements for each process
- Each process declares the max number of instances of each resource that it may need
- Upon receiving a resource request, the deadlock avoidance algorithm checks if the granting of the resource leaves the system in a safe state
- If so, grant the request, otherwise wait until the state of the system changes to a state where the request can be granted safely

How to determine whether a state is safe

- State is cycle free
- Even if there is a cycle, it can still be safe
- So, we use the Banker's Safety Algorithm

Banker's Safety Algorithm

- 5 processes, p_0, p_1, p_2, p_3, p_4
- 3 resources, $a=10, b=5, c=7$

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<u>A B C</u>	<u>A B C</u>	<u>A B C</u>
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- P_1 needs 322, which we can service
- So, we execute it and claim the processes resources back.
- Then we have 532 available

Resource Request Algorithm

Pretend the request is granted and determine if the resulting state is safe using Banker's safety algorithm

- If so, grant request immediately
- Otherwise keep the request pending until state change

Main Memory

02 January 2021 16:03

Objectives

1. How does the OS manage main memory?
 - a. Memory protection
 - b. Address binding
 - c. Logical and physical addresses
2. Explore various ways of allocating memory to processes
 - a. Contiguous memory allocation
 - b. Segmentation
 - c. Paging
 - Page table implementation
 - Page table structure
 - Hierarchical page table
 - Hashed page table
 - Inverted page table

CPU and Memory

- Memory is an array of bytes each having an address
- A program must be taken from disk into memory to be executed
- CPU can read instructions and data from memory and store data into the memory
- CPU fetches the next instruction from address stored in the Program Counter
- Execution of the instruction may involve reading data for memory or writing data into memory
- In either case, memory accessed through an address

Memory Protection

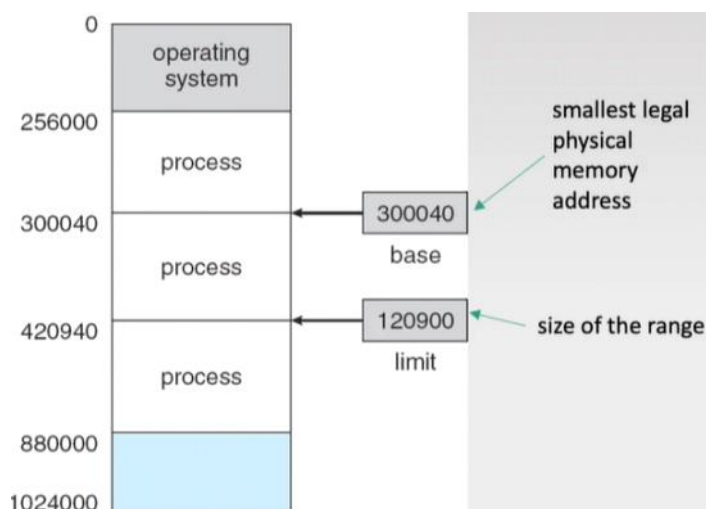
Addresses accessible by a process must be unique to that process – no other process should be able to access the same set of addresses

If not, then processes can write into each other's address space – big security concern

Base and Limit Registers

A pair of base and limit registers define the range of legal addresses

The OS loads these registers when a process is scheduled – allows only the OS to change the base and limit as required



- CPU checks if each address falls within the legal range
- If not, it lets the OS know which takes necessary actions to prevent the memory access

Method assumes contiguous memory allocation, but other methods exist.

Address Binding

A program usually resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.

Depending on memory management, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue.

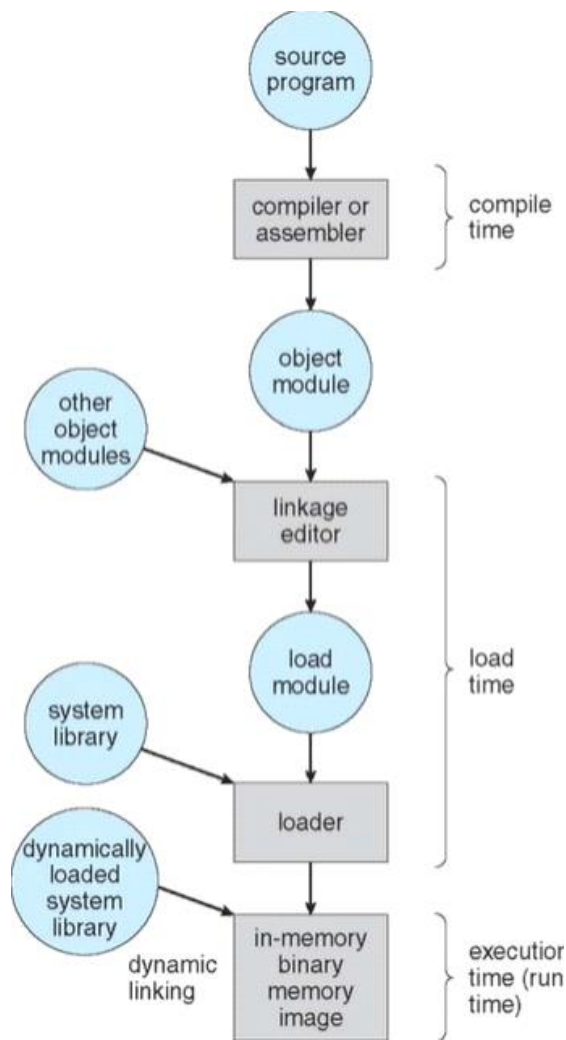
Generally, the user program may go through many steps before being executed. Addresses are represented in different ways at different stages of a program's life
e.g. **variable, count, function** refer to memory addresses that depend on step we're in in the program execution.

When we compile the code, we **bind** the symbolic addresses to relocatable addresses e.g. "14 bytes from the beginning of this module, you will find X"

The linkage editor or loader in turn binds the relocatable addresses to absolute addresses such as 74014. Each binding is a mapping from one address space to another

We can bind instructions and data to memory addresses at any step along the way:

1. **Compile time** – if you know at compile time where the process will reside in memory, then absolute code can be generated. For example if you know that a user process will reside starting at location R, then the generated compiler code will start from that location and extend from there. However, if the location of the process changes, then the starting location will change and it will be necessary to recompile the program.
2. **Load time** – if it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. If the starting address changes between compilation and execution, we need only reload the user code to incorporate the changed value. Final binding is delayed until load time.
3. **Execution time** – If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Most general purpose OS's use this method, but it does require special hardware that we will discuss.



Logical Versus Physical Address Space

An address generated by the CPU is typically called a **logical address**, whereas an address seen by the memory unit, that is one loaded into the MAR, is commonly referred to as a **physical address**.

The *compile-time and load-time address-bindings methods generate identical logical and physical addresses*. However, the execution-time address binding scheme results in different logical and physical addresses.

In this case we usually refer to the logical address as a **virtual address**.

- The set of all logical addresses generated by the program make up the logical address space.
- The set of physical addresses corresponding to these logical addresses is a physical address space

Thus in execution-time binding, we have a different physical and virtual address space.

During run time, the mapping of the physical address space to the logical address space is done by the MMU (Memory Management Unit). We can choose many different methods to achieve this mapping.

A simple example of an MMU scheme would be to have a value of 14000 in the relocation register. Thus, when we try to access logical address 345, the MMU maps that to 14345.

The user program deals with these logical addresses, and so can manipulate 345 as much as it

wants, without knowing that it's actually being remapped to 14345. The Memory Mapping hardware converts these logical addresses into physical addresses outside the program.

Contiguous Memory Allocation

Main memory must accommodate the OS and the user processes. We therefore need to allocate main memory in the most efficient way possible. One method is CMA.

The memory is usually divided into two parts – one for the OS and one for the user processes.

In CMA each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

We can prevent a process from accessing memory that it does not own by combining two ideas – the relocation register and the limit register.

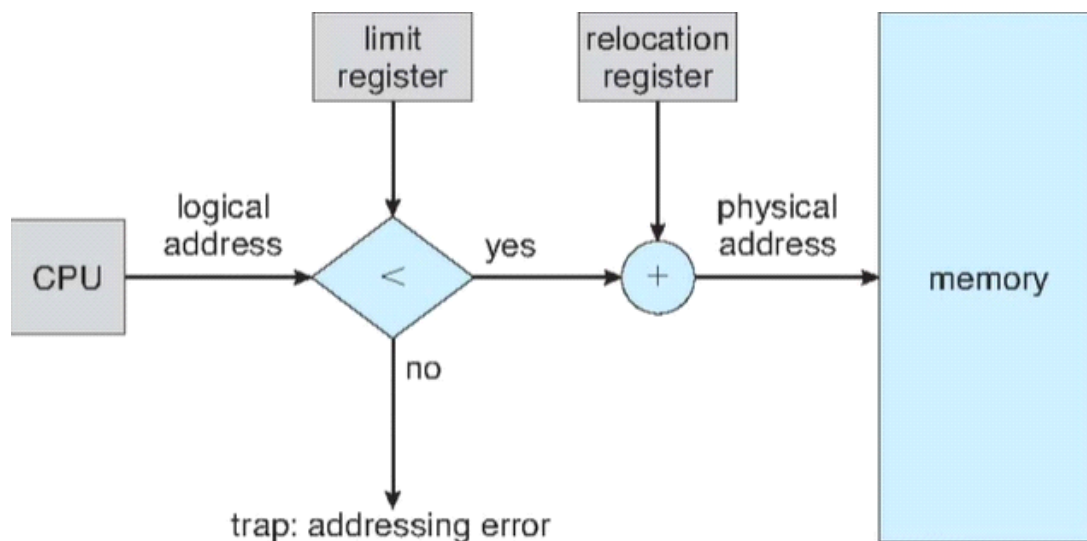
The relocation register contains the value of the smallest address physical address

The limit register contains the range of logical addresses

e.g. relocation = 100040 and limit = 74600. Each logical address must fall within the range specified by the limit register.

The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users programs and data from being modified by this running process.



Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.

In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for the user processes and is considered one large block of available memory, a hole.

At any given time we have an input queue, and a list of available hole sizes. The OS can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied.

The OS then waits until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

1. **First fit** – this allocation method finds the first hole that is big enough to fit the process. We stop searching when we find a hole that is large enough
2. **Best fit** – Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
3. **Worst fit** – allocate the largest hole – we must search the entire list unless it is sorted. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Best-fit and first-fit are better than worst-fit in terms of storage utilisation. First-fit is generally fastest.

Fragmentation

Both first-fit and best-fit strategies for memory allocation suffer from external fragmentation. As processes are loaded and removed, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous – storage is fragmented into a large number of small holes.

50% Rule – given N allocated blocks, another $0.5N$ will be unusable.

If we have a block of 18,723 bytes and we have a process that requests 18,720 bytes, then there will be 3 bytes left over. Keeping track of this small hole costs substantially more than the 3 bytes we're keeping track of.

To avoid this, we break down the memory into fixed size blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation – unused memory internal to a partition.

Solution to External Fragmentation

- **Compaction** – shuffle memory contents to place all free memory together. Not possible sometimes, e.g. when relocation is static and done at assembly or load time. Possible only when relocation is dynamic and done at execution time.
- **Permit logical address space to be non-contiguous** – allows a process to be allocated physical memory wherever such memory is available. Two techniques achieve this solution – segmentation and paging. These techniques can be combined.

Segmentation

The user's view of memory is not the same as the actual physical memory. Segmentation provides a way of mapping the programmer's view to the actual physical memory, so that the system has more freedom to manage memory.

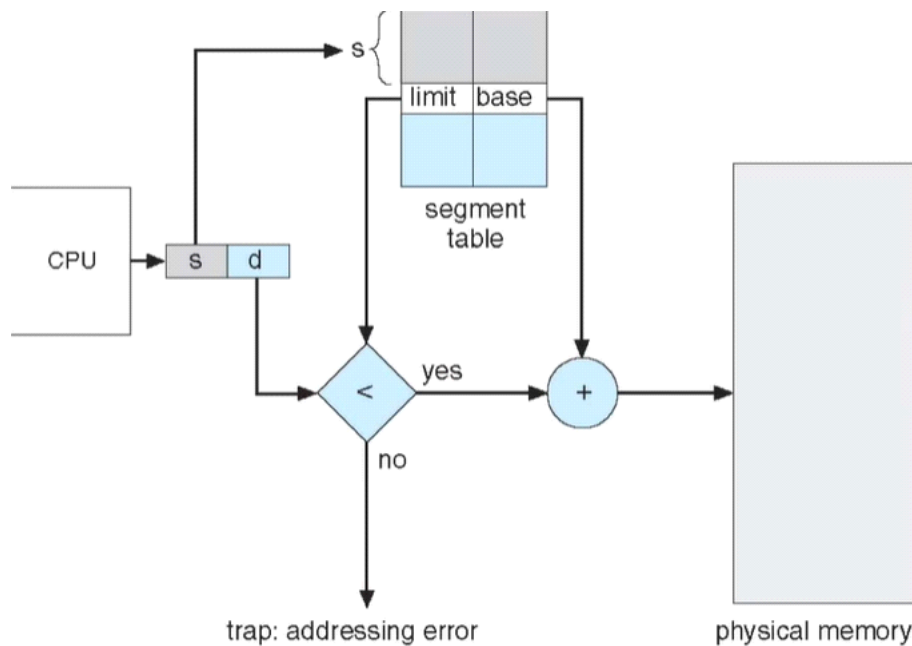
Segmentation is a memory management method that supports the programmers view that

each section of code is stored in its own segment of memory. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities – a segment and an offset.

<segment-number, offset>

A C compiler might create separate segments for the following:

- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library



Paging

Summary of method:

- Divide program into fixed-sized blocks called pages
- Divide physical memory into fixed sized-blocks called frames
- Page size = frame size = e.g. 4kb

Segmentation permits the physical address space of a process to be non-contiguous. Paging is another memory management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

It also solves the problem of fitting memory chunks of varying sizes onto the backing store.

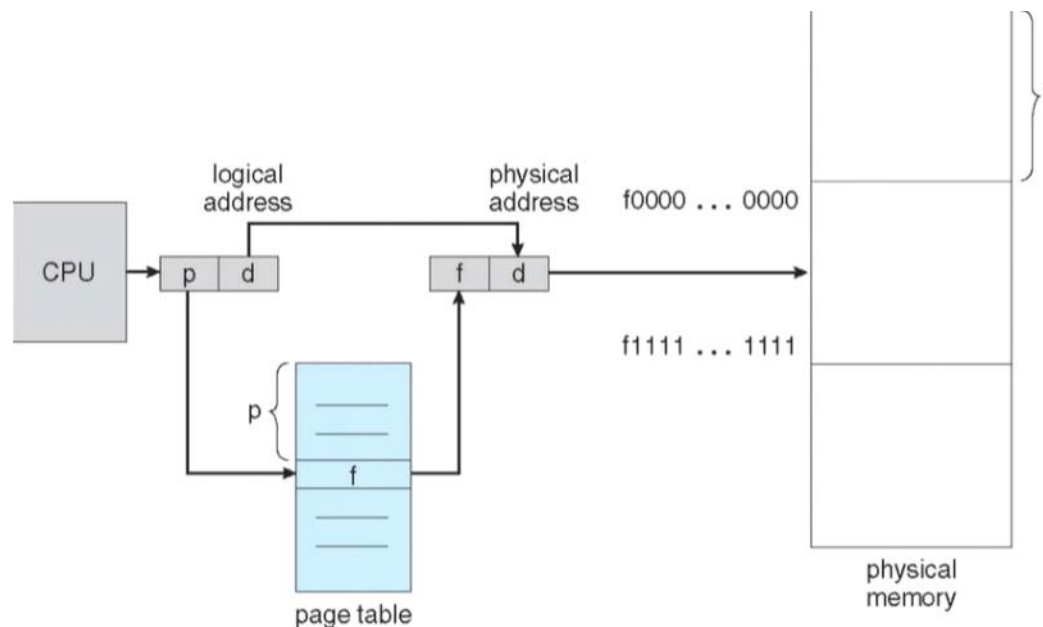
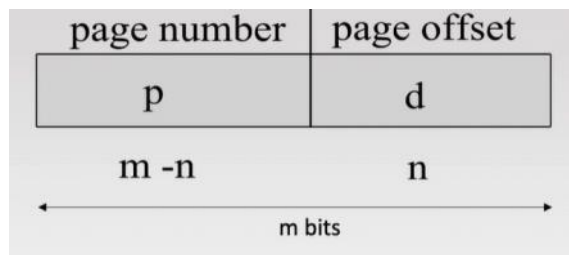
The method involves breaking down the memory into fixed-size blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source.

Each address generated by the CPU is divided into two parts – the **page number** and the **page table**. The page number is used as an index in the page table. The page table contains the base address of each page in physical memory. *This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.*

Page number – used as an index into a page table which contains base address of each page in physical memory

Page offset - combined with base address to define the physical memory address that is sent to

the memory unit



We end up with some internal fragmentation - if the page size is 2048 bytes and the process is 72,766 bytes, then we end up with 35 frames plus 1086 bytes. Thus, we need to allocate 36 bytes and end up with 962 wasted bytes.

Are smaller frames desirable?

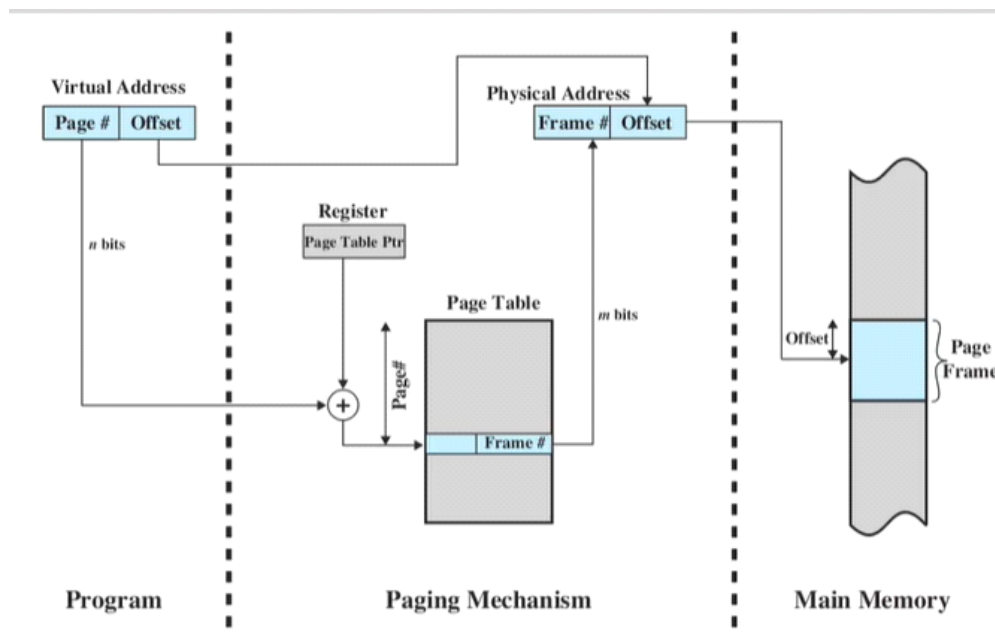
- Smaller frames mean larger page tables
- Page size have grown over time – with growing data sets and main memory

Page Tables

- Remember that the page table stores the mapping between page numbers and the frame numbers.
- OS keeps a page table in the memory for each process

MMU can consist of a set of fast registers to hold page table entries

- These registers are loaded by the OS when a process is scheduled
- Fast registers are of limited size – not suitable for systems with millions of page table entries
- e.g. 32 bit logical addresses, 4KB page size $\Rightarrow 2^{20}$ page table entries



Instead, we can access the page table directly from the memory. We need to store the base address of a table in a register called page table base register (PTBR)

The problem with this approach is that the overhead of memory access doubles – you have to access memory to get the page information before you access memory again

Instead, we use a translation lookaside buffer (TLB)

TLB stores < 256 frequently used page table entries – lookup is very fast

If the page is not found in the TLB, then it's a cache miss. The actual page table has to be found

When the table entry is brought from memory, it evicts an entry of the TLB.

Many cache replacement algorithms can be considered.

- One popular choice is least used (LRU)
- Each algorithm has a corresponding hit ratio.
- The effective access time depends on the hit ration
- $EAT = \text{hit ratio} \times 1 \times \text{memory access} + (1 - \text{hit ration}) \times 2 \times \text{memory access}$

E.g. with a 95% hit ration and a 100ns access time, what is the EAT?

$$0.95 \times 100 + (0.05) \times 200 = 105\text{ns}$$

The TLB can store page table entries for multiple processes, so each entry needs an identifier to uniquely identify the process requesting the TLB search.

This is called an **Address Space Identifier or ASID**. If ASID's match, then the frame number is returned, otherwise the request is considered to be a cache miss. This guarantees memory protection.

Structure of the Page Table

Most processes only use a small number of logical addresses out of the total 2^{20} logical addresses.

A valid-invalid bit is used for each page table entry to indicate if there is a physical memory frame corresponding to a page number

- This bit is 1 when there is no physical frame corresponding to a page

Most of the 2^{20} entries would be invalid, yet they would occupy memory space

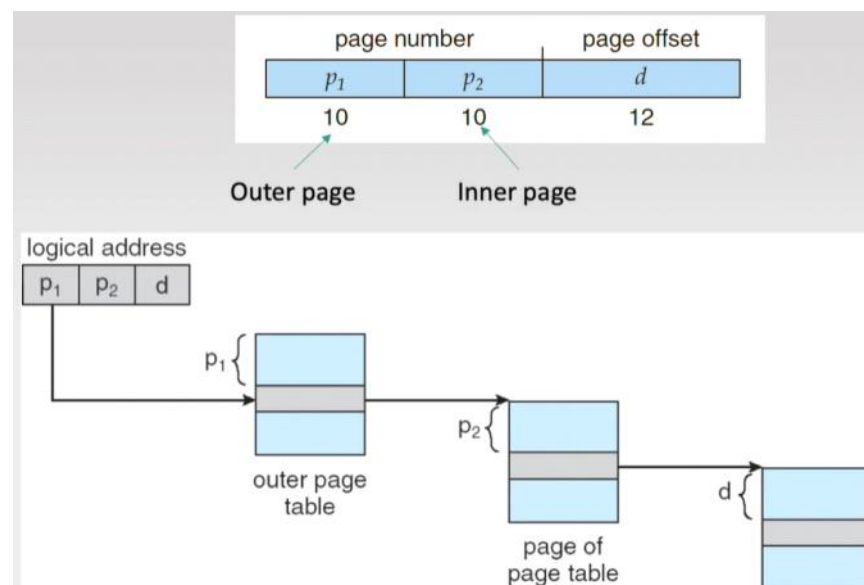
It would be better to group together invalid entries and mark them as invalid memory spaces.

Hierarchical Page Tables

In this method, we page the paging table. Pageception. We divide the page table into pages and store each page in a frame in the memory

The mapping between pages and frame numbers of the paged paging table can be stored in an outer page table. If the outer page table is still larger than one page, we can further divide it.

The advantage is, we have multiple smaller page tables. If a block of logical addresses are not used, we can set the invalid bit and ignore those bits.

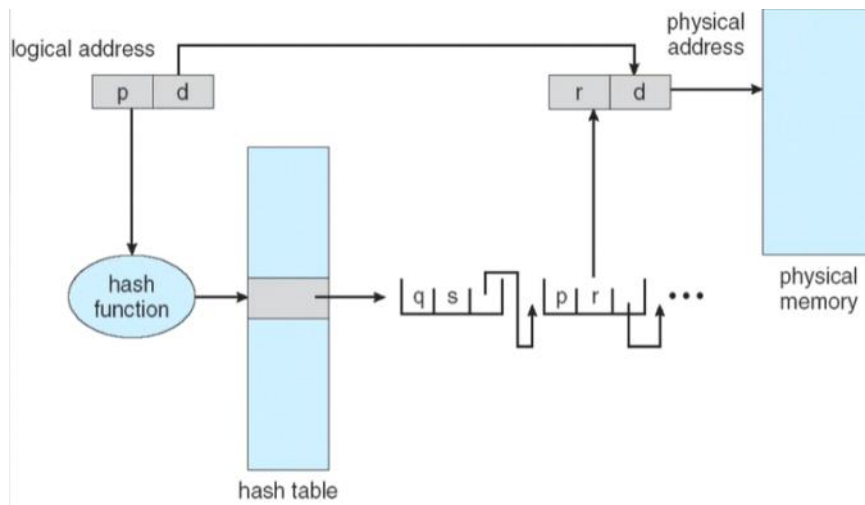


Hash Page Tables

A hashed page table has a hash value that is the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location. Each element consists of three fields – the virtual page number, the value of the mapped page frame, and a pointer to the next element in the linked list.

Algorithm

- Each virtual address is hashed into the hash table
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is match, the corresponding page frame in field 2 is used to form the desired physical address
- If there is no match, subsequent sentries in the linked list are searched for a matching virtual page number.



Inverted Page Tables

A page table has one entry for each page that the process is using, or one slot for each virtual address, regardless of the latter's validity.

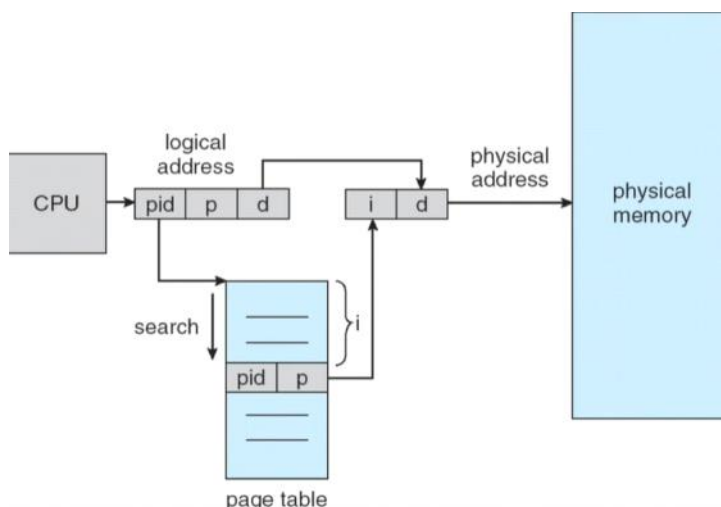
This table representation is a natural one, since processes reference pages through the pages' virtual addresses.

Since the table is sorted by virtual address, the OS is able to calculate where in the table the associated physical address entry is located and use that value.

One of the drawbacks is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

We can solve this with an inverted page table.

It has once entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus only one page table is in the system and it has only one entry for each page of physical memory.



Each virtual address consists of the following:

<process-id, page-number, offset>

Process-id = assumes the role of the address-space identifier

When a memory access is attempted, the process-id is searched in the inverted page table. If a match is found, then the physical address <I, offset> is generated. Otherwise, it is an illegal access.

This scheme decreases the amount of memory needed to store each page table, but increases the amount of time needed to search the table when a page reference occurs.

Because the inverted page table is sorted by physical addresses, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long.

Introduction to Computer Networks

06 November 2020 09:48

Objectives

- Networks and their functions
- Network delays and loss
- Packet switching vs Circuit switching
- Layered structure of the internet

What is a Computer Network?

A network of inter-connected computing devices which enable processes running on different devices to communicate

Components

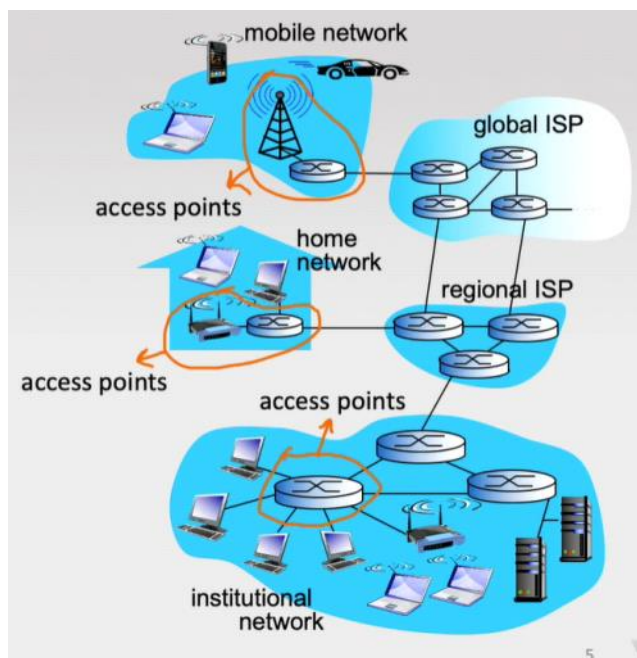
- Network edge consists of end hosts running apps
- Packet switches help forward data packets – switches and routers
- Communication links carry data packets between network devices e.g. fibre, copper, radio, satellite

End hosts connect to the internet through access points

AP's are provided by internet service providers

ISP's are organized in hierarchical structures

Internet is the network of ISP's



End Hosts

Run app processes which generate messages

Breaks down application messages into smaller chunks called packets

Adds additional information (headers) like port number and IP dest. We need port because end hosts run many processes so need to know which process the packet is for

Sends bits of a physical medium

If needed, provides reliable and orderly delivery of packets

Controls rate of transmission

Access Points

End points connect to access points

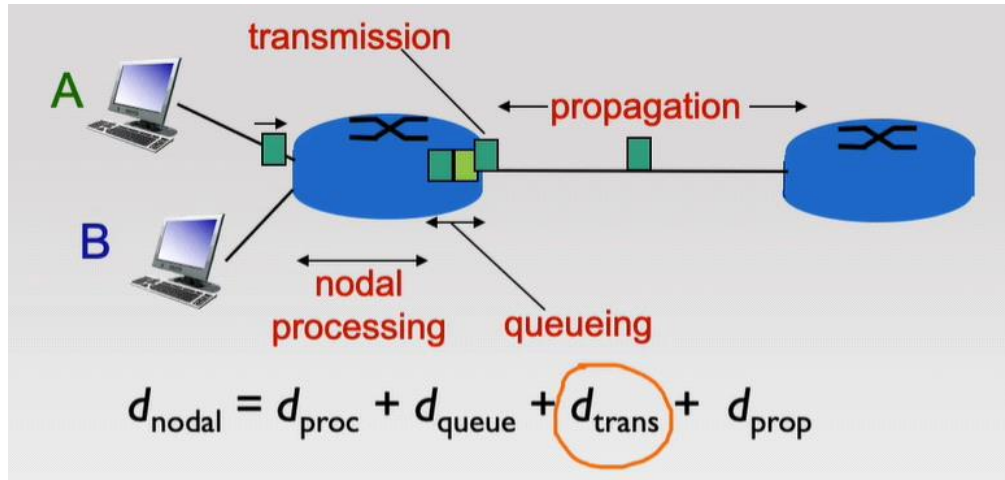
Typically ethernet and WIFI to connect to routers

Network Core

Packet switches run routing algorithms which compute the next hop router for the next hop router

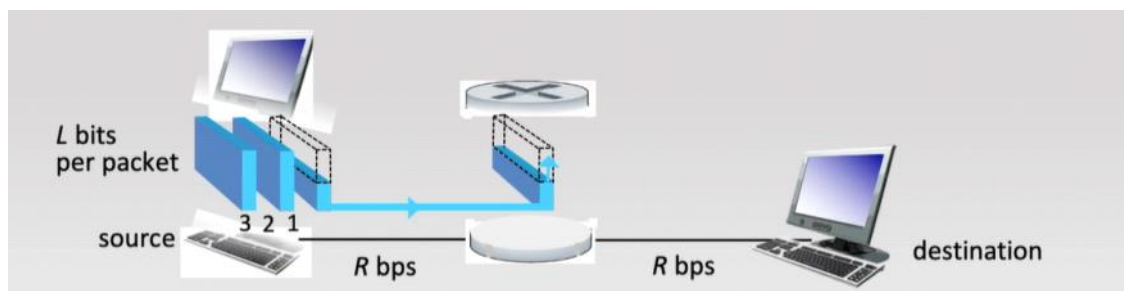
Forwarding: once a packet arrives it is forwarded to the appropriate link

4 Sources of Packet Delay



Transmission Delay - Store-and-Forward Principle

Entire packets must arrive at router before it can be transmitted on next link



Thus it takes L/R seconds to transmit L -bit packets into a link at R bps

Queuing Delay

- If arrival rate to links exceeds transmission rate of links for a period of time then packets will queue, waiting to be transmitted on link
- Packets can be dropped if memory fills up

Propagation Delay

Length of physical link

Propagation speed in medium

Nodal Processing

Checking bit errors

Determining output link
Typically < msec

Throughput

We can talk about data transmission speed in terms of throughput
Specific to a flow or communicating pair
Rate at which bits are transferred from src to dest in a given time window

Protocol

Communicating nodes must agree on certain rules
A protocol defines rules for communication – what to do if a message is received and sequence and format of messages

Protocols define the format and order of messages sent and received among network entities, and actions taken on message transmission and receipt

Network protocols can be implemented as software or as hardware

E.g. routers run protocols to forward message e.g. IP implemented as software
Network interface cards implement hardware protocols to send bits through physical mediums

Packet Switching

The internet uses packet switching
Different flows can share resources along their routes

If one flow is not using any of the shared links then the other flow can use it
Flow can change routes

However, due to the way resources are shared, packets may have to wait in a queue and can even be lost in buffer overflows

Hence we can use circuit switching in some cases as there is a guaranteed rate and no losses.

Circuit Switching

Was used in telephone networks
A circuit is reserved for each flow for the entire call duration

- Flows do not share resources
- If one flow is not using its assigned circuit during the call, it cannot be used by another flow
- Not ideal for internet traffic which is bursty in nature

Layering

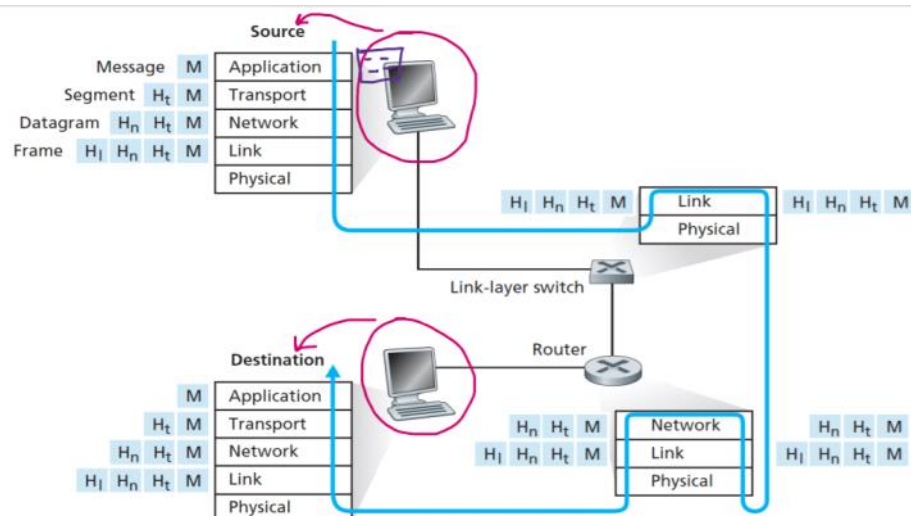
- Network devices perform complex functions
- It is better to divide these functions into layers
- Each layer performs a subset of functions
- Layer N uses the services on the layer below, N-1
- Similarly, the layer above N can use the services of N

The layering approach lets system designers change layers, adding or changing services without affecting other layers

Internet Protocol Stack

The internet protocol stack has 5 layers containing all possible functions – all layers may not be present in a network device

Application Layer Generate data to be communicated over the internet. HTTP SMTP DNS
Transport Layer Packetize the data, add port no., add sequencing and error correcting info TCP UDP
Network Layer Add source and destination IP addresses, Run routing algorithm IP Routing protocols
Link Layer Add source and destination MAC addresses, Pass frames onto NIC drivers Ethernet WiFi
Physical Layer Send individual bits through the physical communication link Separate protocol for each physical medium: co-axial cable, WiFi etc.



Selected Networking Topics

05 April 2021 10:43

MAC Addresses

It is not hosts and routers that have link-layer addresses, but rather their adapters (network interfaces).

A host or router with multiple network interfaces will therefore have multiple-link layer addresses.

These link-layer addresses are called MAC addresses.

MAC Addresses are 6 bytes long, giving 2^{48} possible MAC addresses. They are expressed in hexadecimal notation, with each byte of the address expressed as a pair of hexadecimal numbers.

No two adapters have the same address, because the IEEE manages the MAC address space. When a company wants to manufacture adapters, it purchases a chunk of the address space consisting of 2^{24} addresses from IEEE.

When an adapter receives a frame, it checks to see if the MAC address matches. The adapter then extracts the enclosed datagram and passes the datagram up the protocol stack. If there isn't a match, the frame is discarded.

If the sending adapter DOES want all other adapters on the LAN to receive and process the frame, then it sends a special MAC broadcast address into the destination address field of the frame. For LANS with 6 byte addresses, the broadcast address is 48 consecutive 1's. FF-FF-FF-FF-FF-FF.

ARP

Address Resolution Protocol

Because there are both network layer addresses (IP address) and link layer addresses (MAC), there is a need to translate between them. For the internet, this is the job of ARP.

Why do we have both MAC and IP addresses?

- 1) LANs are designed for arbitrary network-layer protocols, not just IP and the internet. If adapters were assigned IP addresses instead of MAC addresses, then adapters would not easily be able to support other network-layer protocols like IPX
- 2) If adapters were to use network layer addresses instead of MAC, the network layer address would have to be stored in the adapter RAM and reconfigured every time the adapter was powered up.

Why not just have no link-layer address in the adapter?

Because then we'd have to pass the data all the way up the layers to the network layer to check for a match, causing every frame sent on the LAN to interrupt the host, instead of just frames intended for that host.

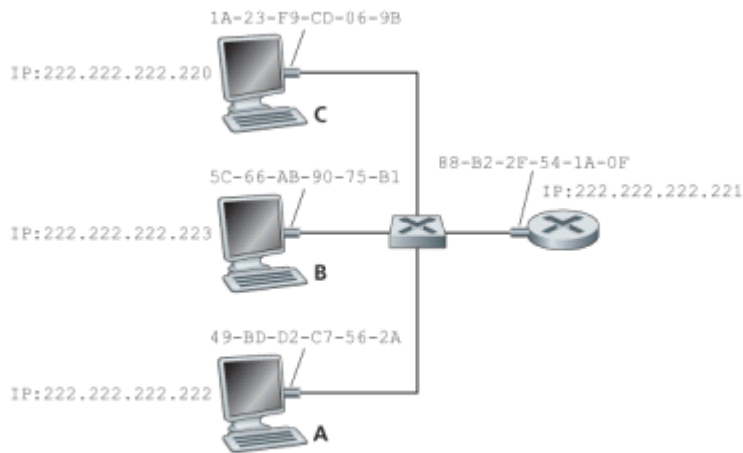


Figure 6.17 Each interface on a LAN has an IP address and a MAC address

How does the sending host determine the MAC address for the destination host with IP address 222.222.222.222?

It uses ARP.

An ARP module in the sending host takes any IP address on the same LAN as input and returns the corresponding MAC address.

ARP resolves IP addresses only for hosts and router interfaces on the same subnet, unlike DNS which resolves host names for hosts anywhere on the internet.

Application Layer

06 November 2020 10:59

Objectives

1. What is a network app
2. How network applications can communicate with other layers
3. Addressing app process running end hosts
4. Transport layer services available to an application process
5. Building network applications: Socket programming

What is a Network Application?

Processes running on different host machines can communicate by sending message over a network
The client process initiates the request for information, and is served by the server process

Developing a Network Application

Dev must develop both client side and the serve side of the program
Developing one side of the program is enough for applications specified by standard protocols

Sockets

User processes constitute the application process
Layers below the application layer are controlled by the kernel

The socket lies between the application and the kernel. **They are software doors between the transport layer and the application layer**

The application layer writes messages into the socket – it's like an API between the application and network

Creating, reading and writing to sockets is done with system call

Messages need to be addressed to the correct process running within the correct end host

Any internet device can be identified by IP addresses.

- Processes can be identified by port numbers
- Port numbers are 16 bit numbers
- Port numbers 0-1023 are reserved for well-known network applications
- Port numbers above 1023 can be used for application programs

Transport Layer Services

Application processes use transport layer services

All transport layer protocols offer some basic services - packetization, addressing, sequencing, error correcting bits

Additional services are sometimes needed – you have to choose the transport layer depending on what you need

TCP and UDP are the main protocols that can be chosen

TCP

Reliable in order data transfer - packets can be lost or arrive out of order but TCP ensures that all packets sent are received in-order

Connection oriented service – setup required between client and server processes before they start transferring data – TCP handshake

- Syn bit sent to server, carries no data but is used as a handshake
- When received, server allocates resources like a port
- The server sends back a syn-ack packet to acknowledge sync
- The client then sends the data back to the server

UDP

UDP provides no guarantees on data transfer

Best effort service – packetize data and sends it to the network

No effort is made to recover losses

UDP has less overhead so is very fast – no connection setup needed

Headers are smaller

Building Network Applications

Goal = learn how to build client/server applications that communicate using sockets

Socket programming creates sockets, reads from and writes to sockets through system calls

Two socket types – connectionless socket and connection oriented sockets

HTTP

Web browsers communicate with web servers using HTTP as the application layer protocol – HTTP is specified by IETF

HTTP uses TCP on port 80

An HTTP request returns a web page from the server – this consists of objects. An object is nothing but a file, e.g. a HTML file, JPEG etc, that is stored on the web server

Each object is addressable by a URL with a host name and a path name

How HTTP Works

HTTP uses TCP – the server runs the server process on port 80

The client initiates a TCP connection to the server, on port 80

TCP connection is established after a TCP handshake which is a 3 way process starting with a SYN bit from the client, followed by a SYN ACK packet from the server, followed by an ACK bit from the server

Once this exchange is complete, the connection is closed

Non-Persistent HTTP Connections (v1.0)

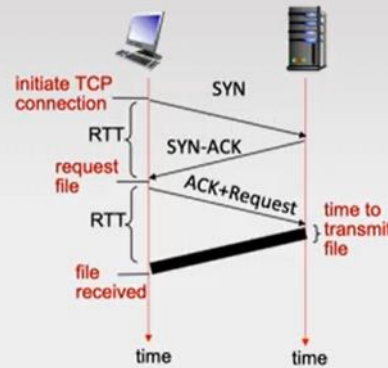
Each object is obtained over a separate TCP connection – downloading multiple objects will require multiple TCP connections

- 1) SYN bit, server responds with SYN-ACK
- 2) client sends ACK and adds request for the base HTML file
- 3) The server establishes the connection and responds with the base HTML file
- 4) HTTP server closes TCP connection
- 5) Client receives the HTML file and examines it to find the 10 other referenced objects

RTT: Round trip time.

HTTP response time:

- 1 RTT to initiate TCP connection.
- 1 RTT to send HTTP request and receive first few bytes of the requested object.
- file transmission time
- response time = $2\text{RTT} + \text{file transmission time}$



Problems with Non-Persistent HTTP

- Each object requires at least 2RTT to be downloaded
- Each TCP connection the OS of the server has to allocate some resources. If there are many objects, then this can be a problem

Persistent HTTP Connections (v1.1)

Multiple objects can be downloaded over a single TCP connection

- Server leaves the connection open after sending response
- Subsequent HTTP messages are exchanged over the open connection
- Client sends requests back to back as soon as it encounters a referenced object
- All objects can be downloaded in $2\text{RTT} + \text{total data transfer time}$

Cookies

An HTTP server is stateless. This simplifies server design and has permitted engineers to develop high performance Web Servers that can handle thousands of simultaneous TCP connections.

However, we often want a site to identify users, to server content as a function of the user identity.

We do this with cookies, which allow sites to keep track of users.

There are 4 components:

- 1) Cookie header in the HTTP response message
- 2) A cookie header line in the HTTP request message
- 3) A cookie file kept on the users end system and managed by the users browser
- 4) A back-end database at the web site

Example of Cookies in Action

- Susan connects to amazon.com
- Server creates a unique ID number and creates an entry in its database, indexed by ID number
- The Amazon web server then responds to Susan's browser, including in the HTTP response a set-cookie header which contains the ID number: `set-cookie: 1678`
- When Susan's browser receives the HTTP response message, it sees set-cookie
- The browser then appends a line to the special cookie file that it manages.
- This line includes the hostname of the server and the ID number in the set-cookie header
- Note that the cookie file already has an entry for other sites that Susan uses.
- As Susan uses Amazon, each time she requests a page, her browser consults the cookie file, extracts her ID, and puts a cookie header, and puts a cookie header line that includes the ID number in the HTTP request
- Specifically, each of her HTTP requests to the Amazon server include `cookie: 1678`
- Amazon is able to track Susan's activity at the site

- Although the server does not know Susan's name, it knows exactly what pages user 1678 has visited, in which order, and when.
- Amazon uses cookies to provide its shopping cart service - maintaining a list of all the items that Susan has added, despite her not being logged in.
- When Susan signs up for Amazon, they can then associate an account with the cookie number, and fill it with all the information she requested in the past

Web Caches

A web cache, or a proxy server, is a network entity that satisfies HTTP requests on behalf of an origin server. The web cache has its own disk storage and keeps copies of recently requested objects in this storage.

A user's browser can be configured so that all of the user's HTTP requests are first directed to the cache.

- 1) Browser establishes TCP connection to the web cache and sends an HTTP request for the object to the web cache
- 2) The web cache checks to see if it has a copy of the object stored locally. If it does, the web cache returns the object within an HTTP response message to the client browser
- 3) If the cache does not have the object, open a TCP connection to the origin server. Then, it sends an HTTP request for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the web cache
- 4) When the web cache receives the object, it stores a copy in local storage, then sends a copy within an HTTP response message to the client browser, over the existing TCP connection.

Why use a web cache?

They reduce the effect of bottlenecks

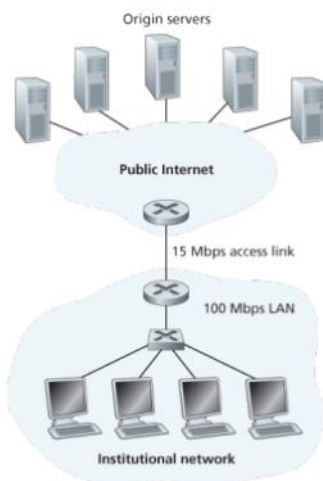


Figure 2.12 Bottleneck between an institutional network and the Internet

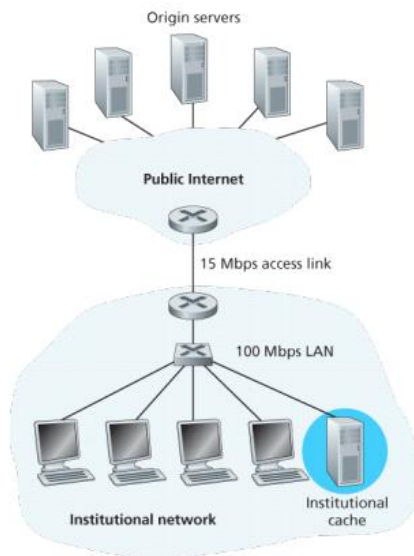


Figure 2.13 Adding a cache to the institutional network

Conditional GET

Caching can reduce user-perceived response times, but introduces a new problem - the copy of an object residing in the cache may be stale.

HTTP has a mechanism called conditional GET, that allows a cache to verify that its objects are valid and up to date.

An HTTP request is a conditional GET if:

- 1) The request message uses the GET method and
- 2) The request message includes an if-modified-since header line

Example

Cache sends a request message to a web server

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
```

Web server sends response message with the requested object to the cache

```
HTTP/1.1 200 OK
Date: Sat, 3 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Wed, 9 Sep 2015 09:23:24
Content-Type: image/gif

(data data data data data ...)
```

Cache forwards the object to the request browser, but also caches it locally. The cache stores the last-modified date along with the object.

A week later, another browser requests the same object via the cache, and the object is still there. Since this object may have been modified, the cache performs a check by issuing a conditional GET.

Specifically, it sends:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
If-modified-since: Wed, 9 Sep 2015 09:23:24
```

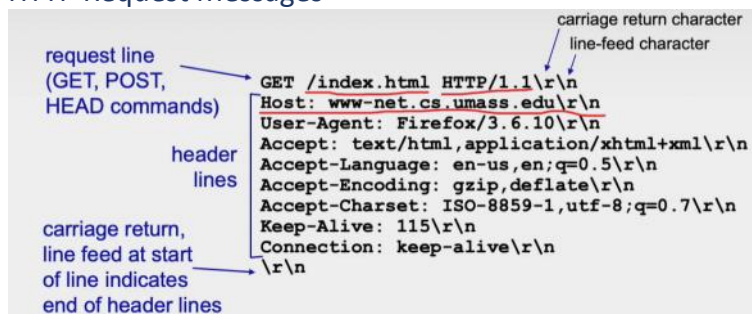
The origin server responds with

```
HTTP/1.1 304 Not Modified
Date: Sat, 10 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)

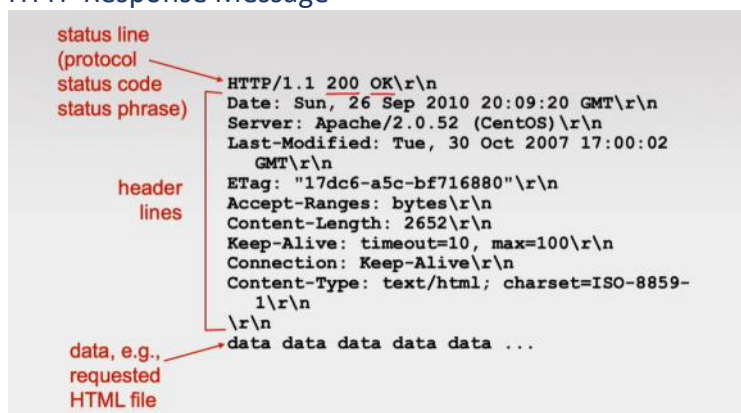
(empty entity body)
```

The web server responds with 304 Not Modified and an empty body.

HTTP Request Messages



HTTP Response Message



We can use web caches to speed up response time – this is because the cache is closer to the client and so the full journey to the main server doesn't have to be made

Transport Layer

02 January 2021 11:00

Objectives

1. What is the transport layer of the internet?
2. Basic services provided by any transport protocol
3. Details of TCP services:
 - a. Reliable data transfer
 - b. Flow control
 - c. Congestion control

What is the Transport Layer?

Transport layer services and protocols provide *logical communication between application processes* running on different hosts.

They abstract the details of the lower layers away. The communicating process on the sending side transmits the data to the transport layer, and the transport layer then handles the communication from there.

On the sending side, the transport layer *breaks the message into segments, adds headers, and passes the packets to the network layer.*

On the receiving side, *it reassembles segments into messages, and passes to the application layer*

TCP & UDP Evaluation

UDP

UDP provides bare minimum services

- Packetization
- Adds headers
- Sends to network layer

If packets are lost or sent out of order, UDP makes no efforts to fix this.

UDP is connection-less – each UDP segment is treated independently.

UDP senders can send at any rate they wish – there is no congestion control

UDP is used because it has less overhead and it is fast

- no connection establishment
- Smaller header size because doesn't need to deliver packets in correct order
- No congestion control – UDP can blast as fast as it wants
- For video streaming, it is not important to be lossless, so UDP is useful

If UDP users want reliable data transfer, they have to add it themselves into the app later

TCP

- Reliable data transfer – recovers losses, re-orders out of order packets
- Flow control – matches the sending speed of the sender to the reading speed of the receiver
- Congestion control – controls the sending rate according to perceived network congestion

Within the internet packets can be lost, corrupted or arrive out of order

- TCP ensures none of this is perceived by the app process
- Hence, TCP has to enhance the unreliable network service

Reliable Data Transfer

- Bit errors can happen because of electrical noise
- Data can get lost in buffers at routers

Checksums – include the sum of all 16-bit words in the header

ACKS – indicates if a packet is correctly received at the receiver

Timeout mechanism – sender times out if ack is not received within a timeout interval

Retransmissions – to retransmit lost or corrupted packets – automatic repeat request (ARQ)

Sequence number – to detect duplicates at the receiver – packets can be duplicated if ACK is lost or corrupted

TCP Connection Management

Establishing Connection

Step 1 - Client side TCP first sends a special TCP segment to the server-side TCP. This segment contains no application layer data. One of the flag bits in the segments header, the SYN bit, is set to 1.

Further, the client chooses an initial sequence number and puts this number in the sequence number field of the initial TCP SYN segment. This segment is encapsulated within an IP datagram and sent to the server. Randomising the client_isn properly avoids certain security attacks.

Step 2 - once the IP datagram containing the TCP SYN segment arrives, the server extracts the TCP SYN segment, allocates the TCP buffers and variables to the connection, and sends a connection granted segment to the client TCP. This is where the server becomes vulnerable to a SYN flooding attack. This connection granted segment also contains no application layer data.

It does, however contain a SYN bit set to 1, and an acknowledgement field set to client_isn + 1. Finally, the server chooses its own initial sequence number (server_isn) and puts this value in the sequence number field of the TCP segment header. This connection granted segment says "I received your SYN packet to start a connection, and I agree to establish this connection".

This is referred to as a SYNACK segment

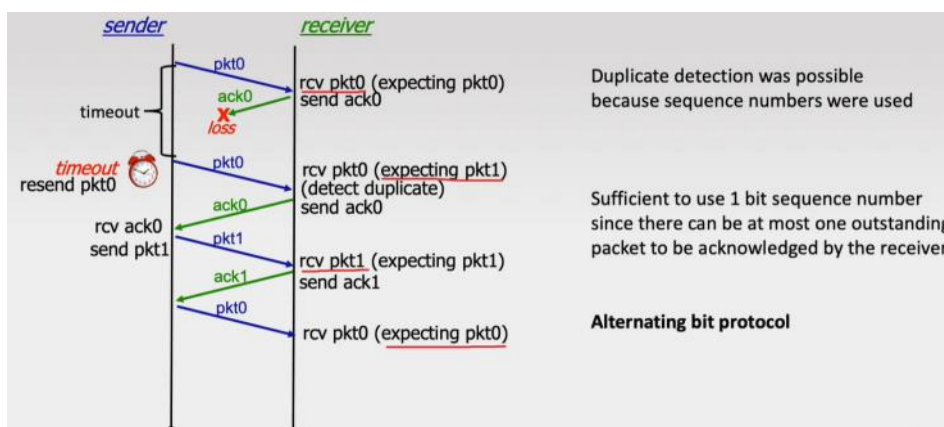
Step 3 - Upon receiving a SYNACK segment, the client also allocates buffers and variables to the connection. The client host then sends the server the last segment, acknowledging the server's connection granted segment (by putting the value of server_isn + 1 in the acknowledgement field of the TCP segment header). The SYN bit is set to 0, since the connection is established.

This third stage of the 3 way handshake may carry client-to-server data in the segment payload.

In future segments, the SYN bit will be set to zero.

Stop and Wait ARQ

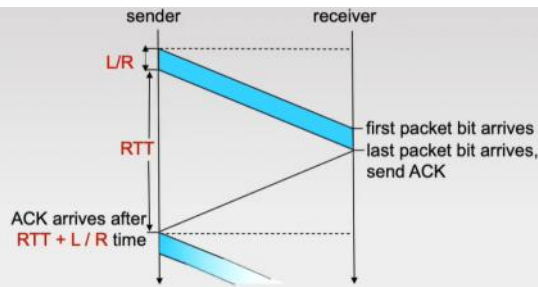
1. Sender sends packet
2. Waits until it receives an ACK
3. If ACK arrives, send next packet
4. Else, time out and retransmit the same packet



Provides reliable data transfer.

Performance of Stop and Wait ARQ

- Suppose we have a 1Gbps link ($R = 10^9 \text{bps}$)
- Packet length $L = 8000$ bits.
- RTT is 30 ms.



$$U = \frac{L/R}{RTT + L/R} = \frac{L}{R \times RTT + L} = 0.00027$$

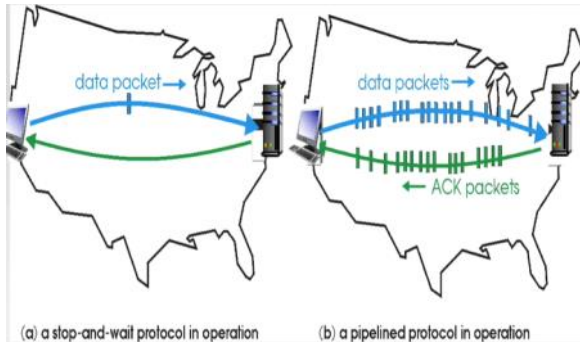
- Sender has to wait until an ACK is received for the previously transmitted packet to transmit the next
- Poor utilisation of the link capacity – poor performance

Bits in the Pipeline

- The receiving process typically has a finite buffer of B bits
- The receiving process may not be reading from its buffer all the time
- Hence, to avoid buffer overflow, the sender should not send more than B bits at a time
- Length of pipeline = $L + R \times \text{RTT}$ bits
- Buffer size = B bits
- Max no of bits without waiting for ACK = $\min(B, L + R \times \text{RTT})$

Pipeline Protocols

- Allow multiple unacknowledged packets in the pipeline
- ACK is sent individually or cumulatively
- Range of sequence numbers must be increased



Generic Pipeline Protocols

There are 2 generic pipeline protocols that are common. TCP uses a combination of both Go-Back-N, and Selective Repeat.

Go-Back-N

- Sender can send up to N packets without ACK
- N = window size, depends on the delay-bandwidth product, receive buffer size and other factors
- The receiver maintains a variable *expectedseqnum* which keeps track of the next expected sequence number to be received
- If the receiver correctly receives packet n and $n = \text{expectedseqnum}$, then it sends $\text{ACK}(n)$ which acknowledges all packets up to and including packet n – cumulative ack – increments *expectedseqnum* by 1
- In all other cases, $n \neq \text{expectedseqnum}$, the receiver discards the incoming packet and sends $\text{ACK}(\text{expected} - 1)$

-
- The diagram illustrates a simple telnet scenario between Host A and Host B. Host A is on the left, and Host B is on the right. The sequence of events is as follows:
- Host A: User types 'C'
 - Host A to Host B: Seq=42, ACK=79, data = 'C'
 - Host B: host ACKs receipt of 'C', echoes back 'C'
 - Host B to Host A: Seq=79, ACK=43, data = 'C'
 - Host A: host ACKs receipt of echoed 'C'
 - Host A to Host B: Seq=43, ACK=80
- simple telnet scenario

Host A

Host B

timeout

Seq=92:99, 8 bytes of data

ACK=100

Seq=92, 8 bytes of data

ignore duplicate

ACK=100

lost ACK scenario

```
sequenceDiagram
    participant A as Host A
    participant B as Host B
    A->>B: Seq=92, 8 bytes of data
    A->>B: Seq=100, 20 bytes of data
    B->>A: ACK=100
    A->>B: Seq=92, 8 bytes of data
    B->>A: ACK=120
```

timeout

delayed ACK

Host A

Host B

timeout

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

ACK=120

Seq=120, 15 bytes of data

cumulative ACK

Host A

Host B

Seq=92:99

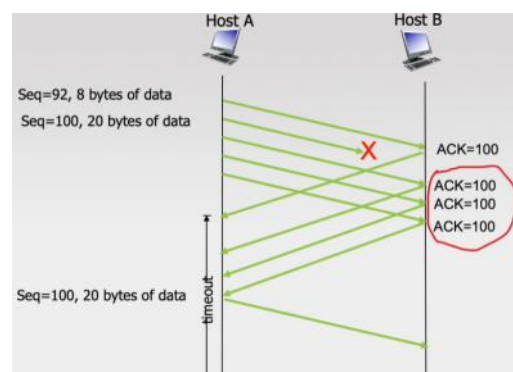
Seq=100:119

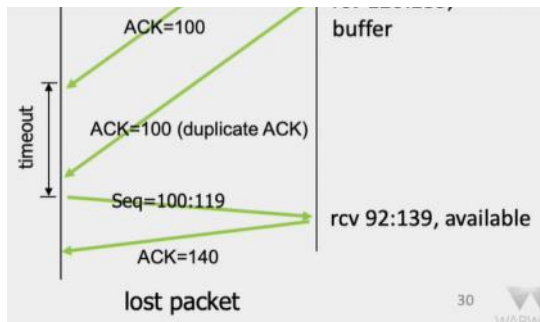
Seq=120:139

ACK=100

rcv 92:99

rcv 120:139, buffer



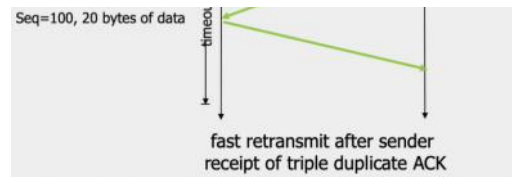


Sender times out waiting for confirmation of 100:119, then retransmits

Timeout period is long, so there is a long delay before resending lost packet

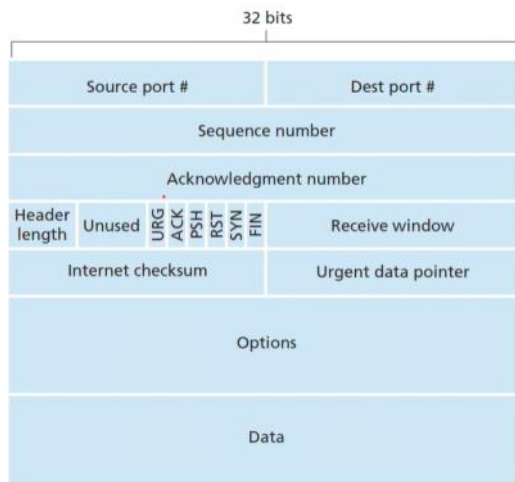
TCP fast retransmit – duplicate ACKs are good indicators of packet loss – they are sent when there is a gap in the received stream of bytes

Upon receiving 3 duplicate ACKs for a segment, TCP sender retransmits that segment without waiting for timeout to occur



TCP fast retransmit. If 3 ACKs are received, then TCP will not wait for the timeout, but rather immediately send the unacknowledged packet. Note that 3 ACKs is much less severe than a timeout, as discussed later

TCP Segment Structure



1. Sequence Number – 32 bit sequence number of the segment indicating the number of the first byte
2. Acknowledgement Number – number of the next byte expected
3. Receive window – used for flow control

Flow Control

- The data in the pipeline should not exceed the receive buffer size
- Otherwise, the receive buffer will overflow and data will be lost

How does the sender know the space in the receivers buffer? We have to use flow control

Receiver advertises free buffer space in the **receive window** field

Sender limits amount of un-ACK'd "in flight" data to the receive window size

Congestion Control

- Reliable data transfer and flow control are directly used by applications
- Congestion control is beneficial for the network as a whole
- TCP Provides congestion control
- Controls the rate of transmission according to the level of perceived congestion
- Occurs at a router when the input rate > output rate

Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of the congestion window and receive window size:

```
LastByteSent-LastByteAcked ≤ min{cwnd, rwnd}
```

Queues in router grow, and the buffer overflows, resulting in lost packets and queue delays. This happens when senders send data too fast

Detecting Congestion

- Detects network congestion through losses and delays
- A TCP sender assumes the network to be congested when a timeout occurs
- Three duplicate ACKs are received

TCP treats these events differently – **A timeout event is taken more seriously** than the reception of 3 duplicate ACKs

Controlling Congestion

- TCP is a window based protocol
- The rate of the sender can be controlled by controlling the size of the send window
- By controlling the window size, we can control the transmission rate

MSS = maximum segment size

Determined by the maximum frame size specified by the link layer protocol

How is the congestion window size varied?

Additive Increase Multiplicative Decrease (**AIMD**)

1. Sender linearly increases the transmission rate until a loss occurs
2. When a loss occurs, decrease the rate by a factor of 2
3. Additive increase – increase cwnd by 1 MSS every RTT until loss
4. Multiplicative decrease – cut cwnd in half every loss

Why AIMD in TCP?

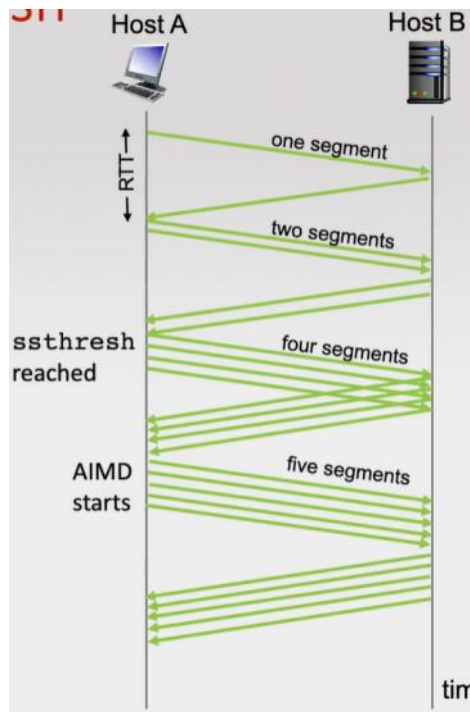
Achieves fair rate allocating among competing flows. Eventually will converge around a fair size for each flow

- Convergence speed of AIMD is low
- To avoid waiting long, TCP uses an initial slow start phase where the window size is increased exponentially fast starting at a small value (slow start) until a threshold is met or a loss is detected
- This ensures that the sender reaches the right operating speed quickly

Reacting to Losses – Timeout and Duplicate ACKs

- Losses are detected through timeouts and 3 duplicate ACKs
- Duplicate ACKs – some segments are lost but some are received – be lenient
- Timeout – no segment is received – take drastic measure

1. **Timeout – drastic action:** ssthresh = 0.5 cwnd, cwnd = 1MSS. After that the sender enters into a slow start phase
2. **3 Duplicate ACKs – take less drastic action** – ssthresh = 0.5 cwnd, cwnd = ½ cwnd and window then grows linearly



An example of slow start in action – MSS grows exponentially until it reaches *sssthresh* (4 bytes here) then grows linearly

Network Layer

03 January 2021 14:04

Objectives:

- Understand the main functions of the network layer
- IP addressing
- Routing

Main Functions

- Moves packets from a source to a destination through intermediate nodes
- Network layer runs in end hosts and routers

One of the main protocols running in the network layer is the IP

- **IP at source:** adds IP header, containing src and dest IP addresses
- **IP at routers:** checks destination IP address of incoming packets to decide the next hop router
- **IP at destination:** receives IP datagram, strips IP header and delivers to transport header

There are other responsibilities of the layer – fragmenting packets and reassembling at destination

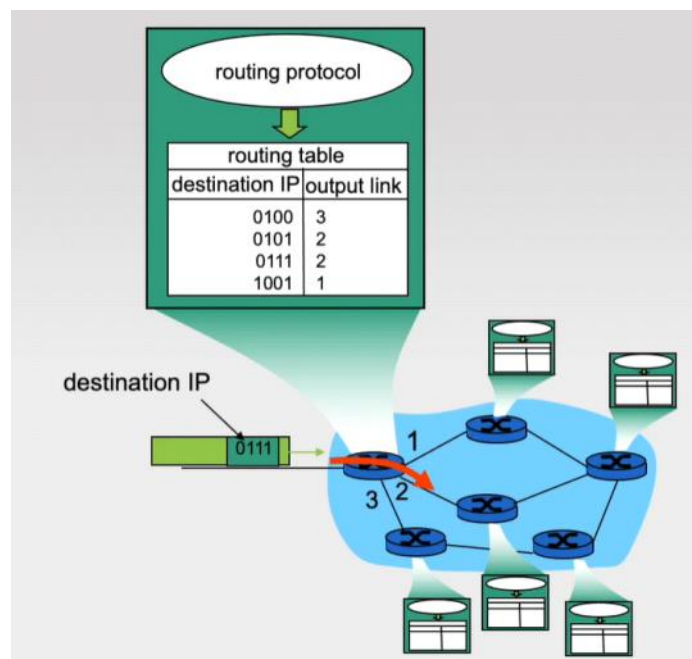
The Network Layer in the routers runs routing algorithms to compute routes

Two Key Functions of Routers

- Forwarding
- Move packets from router's input to appropriate router output

Routing

Constructing routing table – using routing protocol. These run in each routing router



Looks up destination IP in table and decides which output link to use. This table is made using routing protocols

Routing Table

- Impractical to keep a separate entry for each IP
- Increases the size of routing table and lookup time
- Many IPs map to the same outgoing link
- Better if each entry corresponds to a group of IPs and maps that to a single outgoing interface

e.g.

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

Longest Prefix Matching

When looking for forwarding table entry for given destination address, use the longest address prefix that matches destination address.

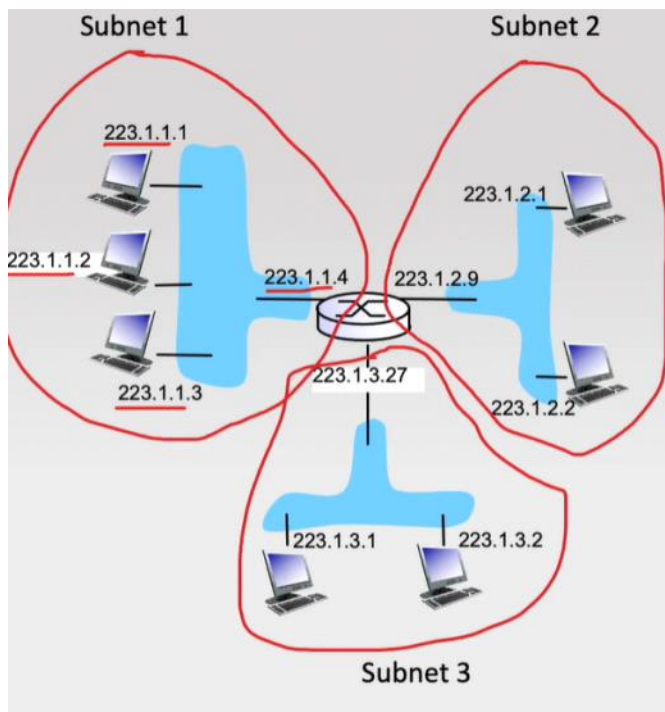
e.g. if 10101100 routes to 1 and 10101011 routes to 2, and we're looking up a destination IP 10101111 then we will choose route 1, since it matches more of the IP than the IP assigned to route 2.

IP Addressing and Subnets

IPv4 addresses are 32 bit addresses which uniquely identify network interfaces

$$223.1.1.1 = \underbrace{11011111}_{223} \underbrace{00000001}_1 \underbrace{00000001}_1 \underbrace{00000001}_1$$

Below we can see 3 different LAN's or Subnets. They have a unique subnet mask which is the prefix. The first 24 bits shows the subnet mask/prefix.



Each subnet mask here has a 24 bit subnet mask.

CIDR (classless inter-domain routing)

Notation: a.b.c.d/x means the first x bits are used as the mask
 e.g. 223.1.3.27/24 means 223.1.3 is the subnet

A sender first checks if the destination IP has the same prefix as its subnet mask. If so, then

- obtain the MAC address of the dest (through ARP)
- Create a link layer frame
- Forward it to the link layer switch

What if the destination doesn't belong to the same subnet?

Forward packet to default gateway

Default gateway will look up the destination in its routing table and forward it to the right outgoing interface

How does a node get an IP address?

- Manually configured
- DHCP – Dynamic Host Configuration Protocol – application layer protocol that dynamically assigns Ip addresses from a server (usually the gateway router) to clients (nodes trying to connect to the subnet)

In either method, the subnet and default gateway must be specified

How does a subnet know which IP addresses to assign?

- The network gets allocated a portion of its provider ISP's address space
- The global authority ICANN is ultimately responsible for allocating IP addresses to ISPs

Network Address Translation

IPv4 addresses are 32 bits, and in short-supply as there are only 4 billion
 ICANN gave out its last IPv4 addresses in 2011

It is not possible for ISP's to provide a unique IP to each device connected to a subnet

Instead a globally unique public IP address is provided to only gateway routers. This is globally unique. These are

called public IPs.

What about other devices?

For these, we allocate private IP addresses

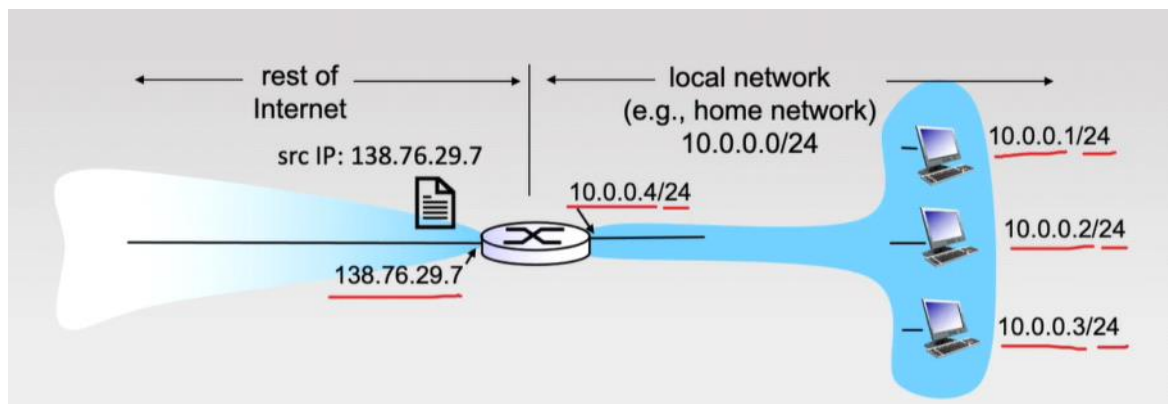
Public and Private IP

Home devices are not publicly visible – they use gateways that are publicly unique.

The following blocks are private IP addresses

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.16.0.0 to 192.168.255.255

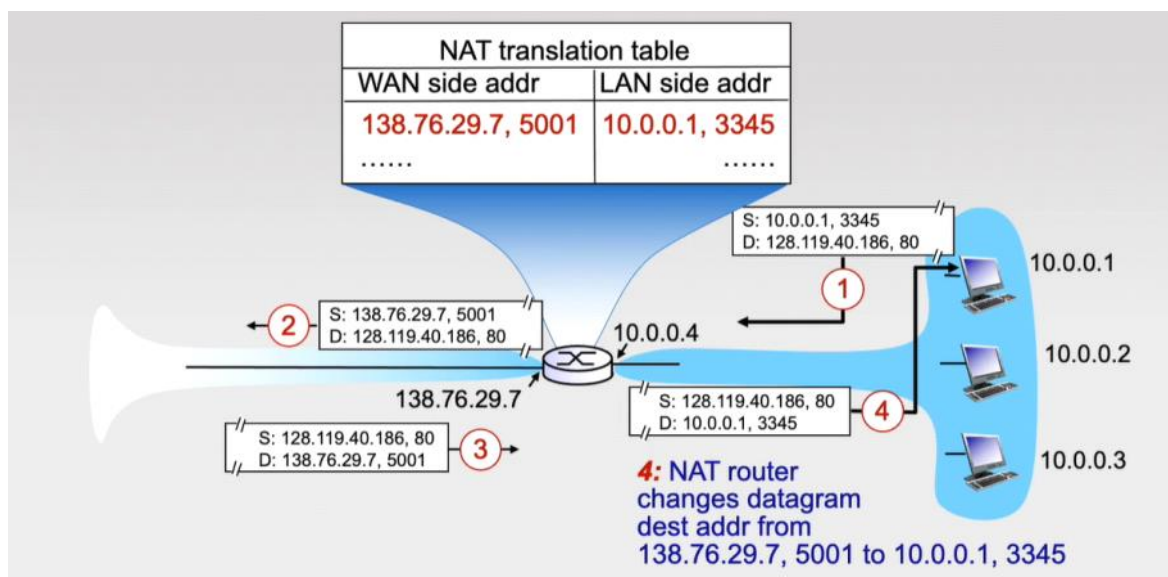
Packets with private IP's can't be sent in the public internet – they will be discarded



It is not sufficient to simply do this.

How will the replies that get sent to the default gateway be distributed to the correct private hosts on the network?

The answer is port numbers.



When the request is sent to the default gateway, the source IP is converted to the IP of the default gateway, with the appended port. This relationship is stored in the NAT, so that when a reply is received it can be looked up and sent to the correct private host

All modern devices now support IPv6

Routing

Each routing protocol implements a routing algorithm that they use to create a routing table

graph: $G = (N, E)$

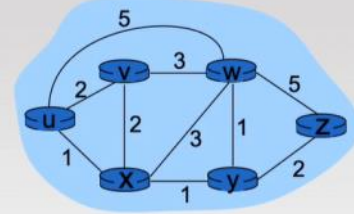
N = set of routers = $\{ u, v, w, x, y, z \}$

E = set of links = $\{ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) \}$

Each edge (x, y) has a cost $c(x, y)$ associated with it,
e.g, $c(u, v) = 2$

$c(x, y) = \infty$ if x and y are not direct neighbours

cost of path $(x_1, x_2, x_3, \dots, x_p) = c(x_1, x_2) + c(x_2, x_3) + \dots + c(x_{p-1}, x_p)$



key question: Given source x find and dest y , what is the least cost path from x to y .
At x we need to know the least cost path to every other node to populate the routing table.

1. Global Routing Algorithms – requires the knowledge of the complete topology at each router including costs – these are link state algorithms
2. Local Routing Algorithms – requires knowledge of only local neighbourhood at each router

Dijkstra's Algorithm - a Link State Algorithm

Computes least cost path from one node to all other nodes

Implemented in OSPF protocol

Each node requires entire topology including the cost of each link

Obtained through broadcasting of link costs or link states

Initialization:

```
→  $N' = \{u\}$  // set of visited nodes, initially contains only the source
for all nodes  $v$ 
  if  $v$  adjacent to  $u$ :
     $D(v) = c(u,v)$  // stores the current estimate of the shortest distance
     $p(v)=u$  // stores the predecessor node of  $v$  along the current shortest path from  $u$  to  $v$ 
  else  $D(v) = \infty$ ,  $p(v)=\text{NULL}$ 
```

Loop:

```
find  $w$  not in  $N'$  such that  $D(w)$  is a minimum
add  $w$  to  $N'$  // least cost path to  $w$  definitively known
for all  $v$  adjacent to  $w$  and not in  $N'$ :
  if  $D(v) > D(w) + c(w,v)$  // update distance to the unvisited neighbor  $v$  of  $w$ 
     $D(v) = D(w) + c(w,v)$  // if the new distance through  $w$  is smaller
     $p(v)=w$ 
```

until all nodes in N'

Distance Vector – a Local Information Algorithm

Based on Bellman Ford equation from dynamic programming

Bellman-Ford equation

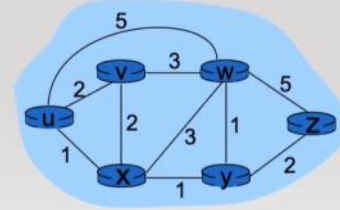
$d_x(y)$ = length of shortest path from x to y

BF equation relates $d_x(y)$ to $d_v(y)$ where $v \in N(x)$ (set of neighbours of x)

BF equation:

$$d_x(y) = \min_{v \in N(x)} \{c(x, v) + d_v(y)\}$$

If v^* minimises the above sum, then v^* is the next-hop node in the shortest path

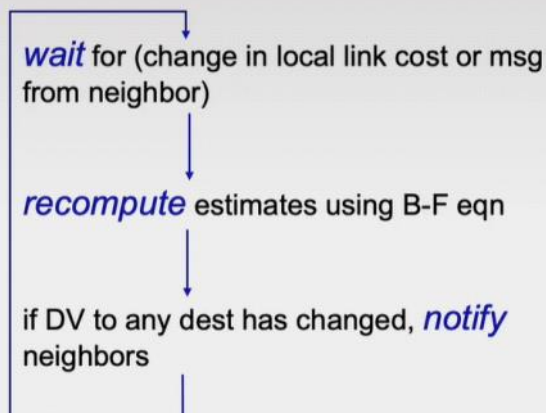


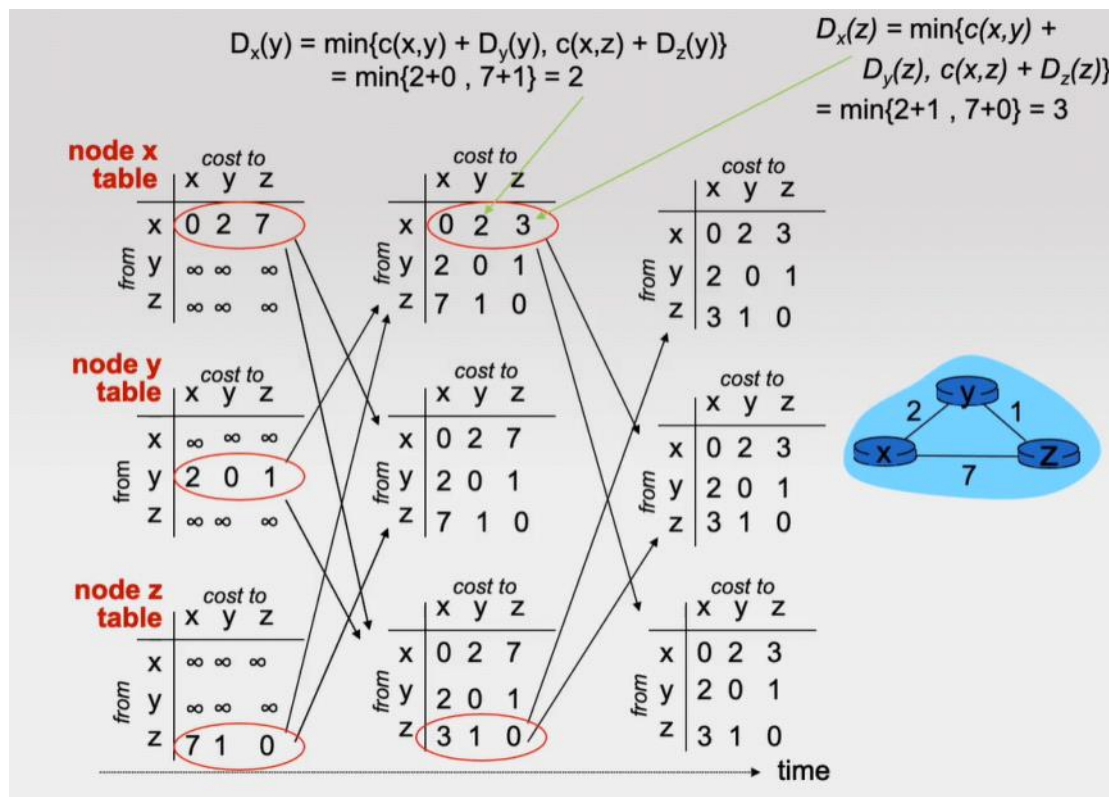
$$\begin{aligned} d_u(z) &= \min \{ c(u,v) + d_v(z), \\ &\quad c(u,x) + d_x(z), \\ &\quad c(u,w) + d_w(z) \} \\ &= \min \{ 2 + 5, \\ &\quad 1 + 3, \\ &\quad 5 + 3 \} = 4 \end{aligned}$$

Next hop node is x

- $D_x(y)$ = current estimate of minimum distance from x to y (different from actual minimum distance $d_x(y)$)
- DV algorithm tries to converge estimates to their actual values
- Each node x maintains distance vector $D_x = [D_x(y): y \in N]$ (vector of current estimates)
- node x performs update $D_x(y) = \min_v \{c(x,v) + D_v(y)\}$
- To perform the update node x needs
 - cost to each neighbor v : $c(x,v)$
 - distance vector of each neighbour v : $D_v = [D_v(y): y \in N]$ (obtained through message passing)
- Whenever any of these is updated, the node **recomputes its distance vector** and **sends the updated distance vector to all its neighbours**

each node:





2020 Paper

26 April 2021 19:03

Question 1

a

i

i=1

child 1 created

Parent prints

Child 1 prints

loops begins with i = 2

Parent creates child 2

Child 1 creates child 3

all 4 print

i = 3

parent creates child 4

child 1 creates child 5

child 2 creates child 6

child 3 creates child 7

all 7 children + parent print

therefore at i=3 we have 8 prints

2^i prints at every level

so $2^{(n+1)} - 2$

-2 since we don't have single print, we start with 2 prints

ii

2^n

b

A PCB is a data structure that stores the information about a process.

The Operating System uses a PCB during an interrupt request, in order to context switch to another task in order to service the interrupt. The state of the current process is saved in the PCB, and then the PCB of the interrupting process is loaded into the CPU.

c

- Heap memory
- Global variables
- Code (text)

d

- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.

e

Memory operations take 40% of execution time
Cache speeds up 70% of memory execution by factor of 4
L2 cache speeds up 15% by factor of 2

100 seconds

60

40 seconds of which are memory operations

$$0.7 * 40 = 28.0$$

$$28 / 4 = 7$$

$$0.15 * 40 = 6.0$$

3

$$100/70 = 1.428$$

Question 2

a

b

c

d

Question 3

a

b

c

d

Question 4

a

F kb from A to B

N links, N - 1 routers

Each has R kb/sec

H kb is added to the payload

Links are congestion free so no queueing delay

propagation and processing delays not relevant

i

Packet size = F + H

Speed = R

it has to do this for N links

Thus propagation delay = $N \times (F + H)/R$

ii

P packets

Each packet size is $F/P+H$ in size

Each packet P has to be sent to the first router individually. After that, the packets are in the pipeline and can all move over the links in unison.

After that, there are $N - 1$ links to travel over

Therefore $(P+N-1) \times ((F/P) + H)/R$

iii

When we remove H from i we get

$$N \times (F/R)$$

When we remove H from ii we get

$$(N - 1) \times \frac{F}{PR} + \frac{F}{R}$$

As such ii is smaller, since $p > 1$, so ii faster.

iv

$$F = 625\text{kb}$$

$$H = 100\text{kb}$$

$$R = 1\text{kb/sec}$$

$$N = 5$$

sub in to equation

$$2500/P + 100P + 1025$$

differentiate to get $100 - 2500/P^2 = 0$

solve for P

$$P = 5$$

b

Each will need $\text{dns_lookup} + 2\text{RTT}_0$ for initial TCP handshake

i

Non-persistent TCP means that we have 2RTT_0 for each object

$$\text{dns_lookup} + 2\text{RTT}_0 + 8 * 2\text{RTT}_0 = 18 \text{RTT}_0$$

ii

$$\text{dns_lookup} + 2\text{RTT}_0 + 2 * (2\text{RTT}_0) = \text{dns_lookup} + 6\text{RTT}_0$$

iii

$$\text{dns_lookup} + 2\text{RTT}_0 + \text{RTT}_0 = \text{dns_lookup} + 3\text{RTT}_0$$

c

d

Question 5

a

b

c

d

Question 6

a

128.119.18.64/26

4 subnets

128 64 32 16 8 4 2 1

10000000.01110111.00010010.01|000000

128.119.18.64/28

128.119.18.80/28

128.119.18.96/28

128.119.18.112/28

b

four devices

24.34.112.235 router address

Network prefix for the home network is 192.168.1.1/24

i

192.168.1.2/24

192.168.1.3/24

192.168.1.4/24

192.168.1.5/24

ii

Two ongoing TCP connections on port 80 (8 total)

Host 128.119.40.86

LAN SIDE ADDRESS	PORT	WAN SIDE ADDRESS	PORT
192.168.1.2/24	82	24.34.112.235	92
192.168.1.3/24	83	24.34.112.235	93
192.168.1.4/24	84	24.34.112.235	94
192.168.1.5/24	85	24.34.112.235	95
192.168.1.2/24	72	24.34.112.235	102
192.168.1.3/24	73	24.34.112.235	103

192.168.1.4/24	74	24.34.112.235	104
192.168.1.5/24	75	24.34.112.235	105

c

2400 bytes datagram into a link that has a MTU of 700 bytes
 IP header length to be 20 bytes
 Datagram has ID 422

i

4 fragments

ii

$2400/4 = 600$ bytes

d