CS162
Operating Systems and Systems Programming

Final Exam Review

December 12, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Final Exam

- Thursday December 15 8-11 am in 155 Dwinelle
- Two double-sided handwritten pages of notes
- Closed book
- Comprehensive
  - All lectures, discussions, projects, readings, handouts,

Topics

- Synchronization
  - Primitives, Deadlock
- Memory management
  - Address translation, Caches, TLBs, Demand Paging
- Distributed Systems
  - Naming, Security, Networking
- Filesystems
  - Disks, Directories
- Transactions

Synchronization Primitives

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads

- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task

- Critical Section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Condition Variables

- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

- Rule: Must hold lock when doing condition variable ops!

Mesa vs. Hoare monitors

- Hoare-style (most textbooks):
  - Signaler gives lock, CPU to waiter; waiter runs immediately
  - Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again

- Mesa-style (most real operating systems):
  - Signaler keeps lock and processor
  - Waiter placed on ready queue with no special priority
  - Practically, need to check condition again after wait

Deadlock

Four requirements for Deadlock

- Mutual exclusion
  - – Only one thread at a time can use a resource.
- Hold and wait
  - – Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - – Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - – There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

Banker's Algorithm for Preventing Deadlock

- Allocate resources dynamically
  - – Evaluate each request and grant if some ordering of threads is still deadlock free afterward
  - – Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}] \le [\text{Avail}]$) for ($[\text{Request}_{\text{node}}] \le [\text{Avail}]$) Grant request if result is deadlock free (conservative!)
  - – Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

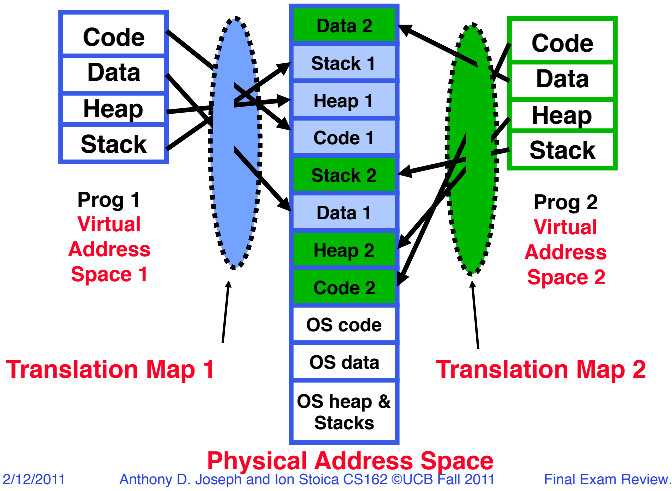
- Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Memory Multiplexing, Address Translation

Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - Can be used to avoid overlap
    - Can be used to give uniform view of memory to programs

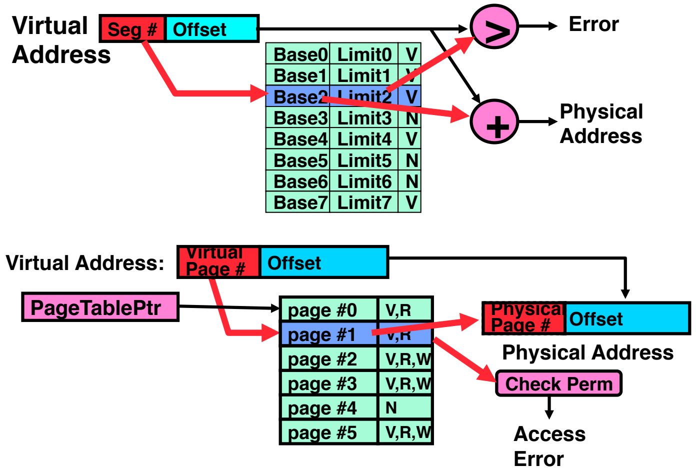
Why Address Translation?

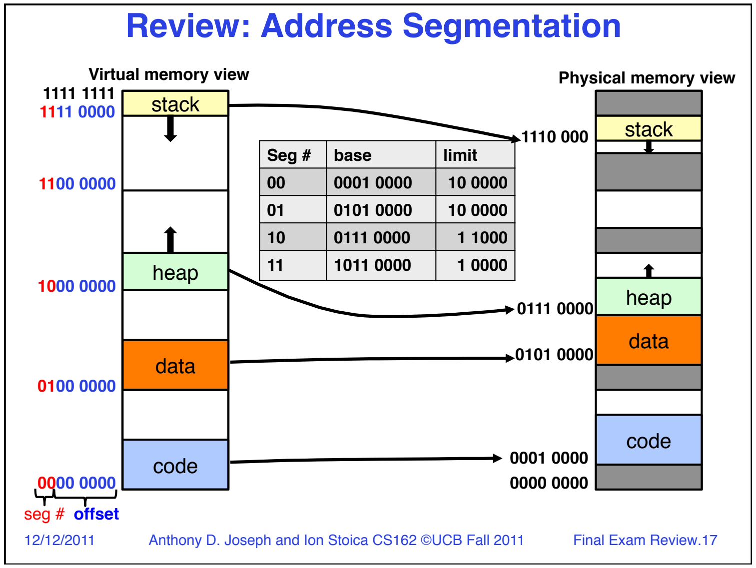


Dual-Mode Operation

- Can an application modify its own translation maps?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To assist with protection, hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
  - User→Kernel: System calls, Traps, or Interrupts

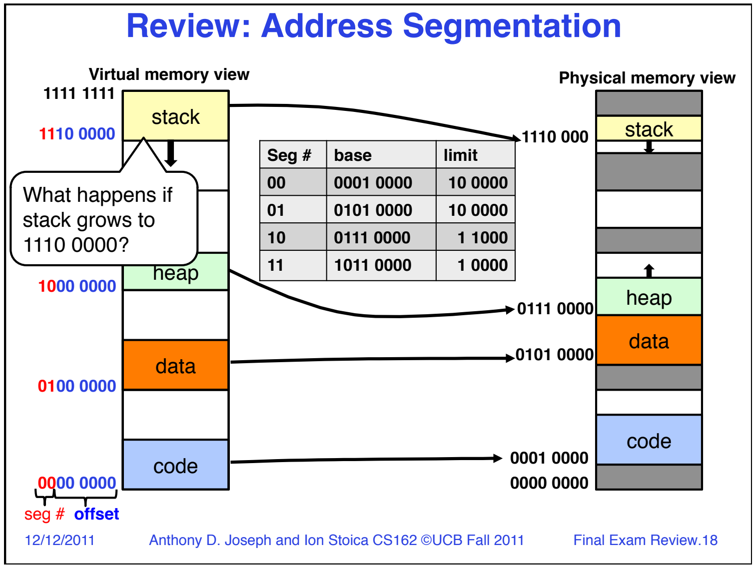
Addr. Translation: Segmentation vs. Paging





Review: Address Segmentation

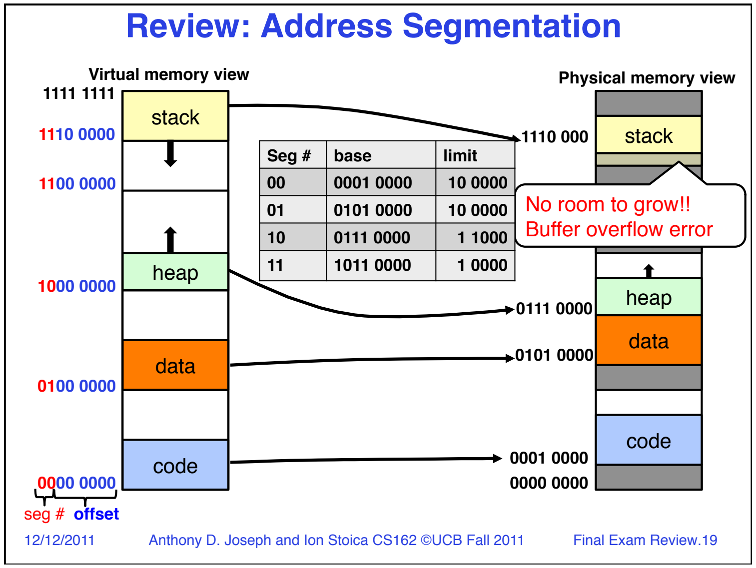
Seg #	base	limit
00	0001 0000	10 0000
01	0101 0000	10 0000
10	0111 0000	1 1000
11	1011 0000	1 0000



Review: Address Segmentation

What happens if stack grows to 1110 0000?

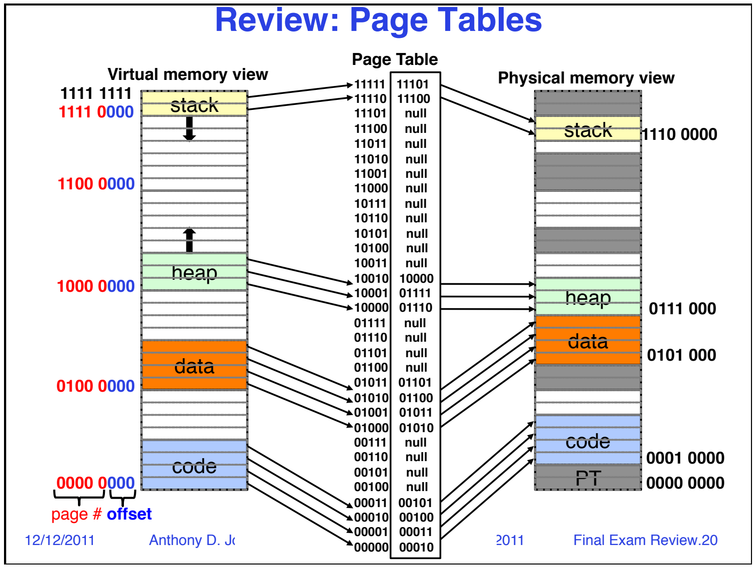
Seg #	base	limit
00	0001 0000	10 0000
01	0101 0000	10 0000
10	0111 0000	1 1000
11	1011 0000	1 0000



Review: Address Segmentation

Seg #	base	limit
00	0001 0000	10 0000
01	0101 0000	10 0000
10	0111 0000	1 1000
11	1011 0000	1 0000

No room to grow!! Buffer overflow error

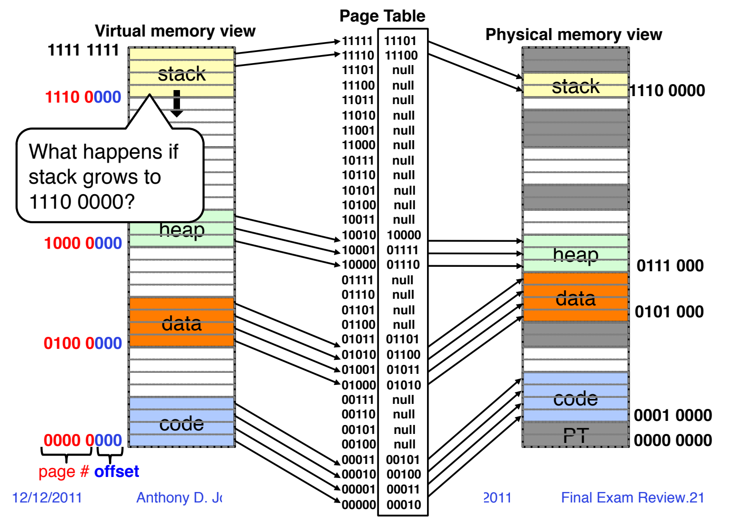


Review: Page Tables

Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Review: Page Tables

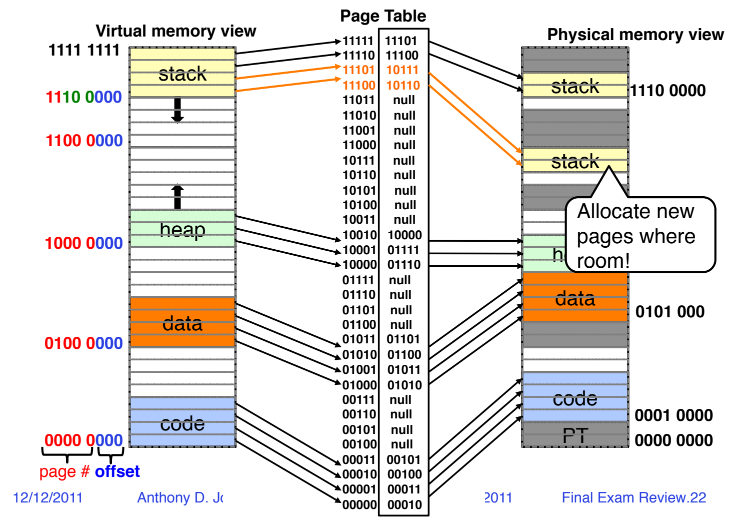


What happens if stack grows to 1110 0000?

Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Review: Page Tables

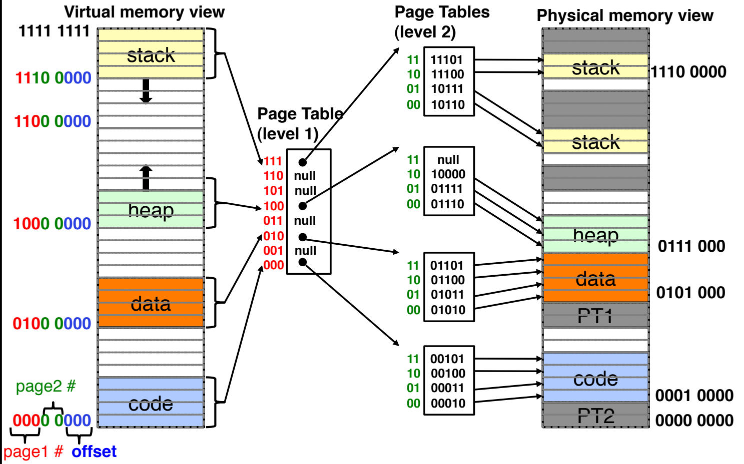


Allocate new pages where room!

Page Table

11111	11101
11110	11100
11101	10111
11100	10110
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Review: Two-Level Page Tables



Page Table (level 1)

111	●
110	null
101	null
100	●
011	null
010	●
001	null
000	●

Page Tables (level 2)

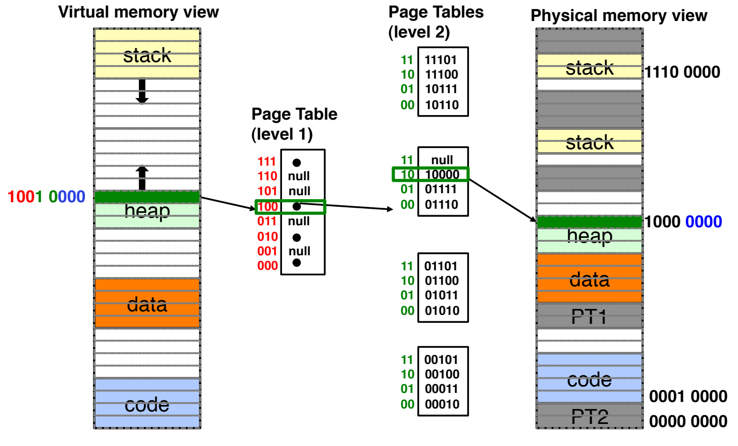
11	11101
10	11100
01	10111
00	10110

11	null
10	10000
01	01111
00	01110

11	01101
10	01100
01	01011
00	01010

11	00101
10	00100
01	00011
00	00010

Review: Two-Level Page Tables



Page Table (level 1)

111	●
110	null
101	null
100	●
011	null
010	●
001	null
000	●

Page Tables (level 2)

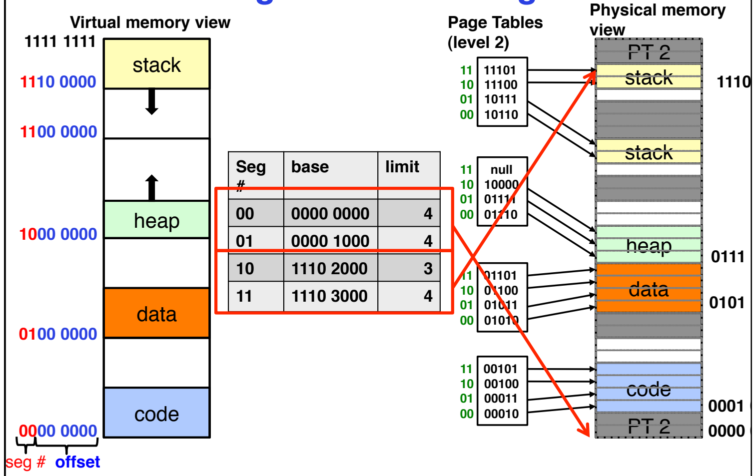
11	11101
10	11100
01	10111
00	10110

11	null
10	10000
01	01111
00	01110

11	01101
10	01100
01	01011
00	01010

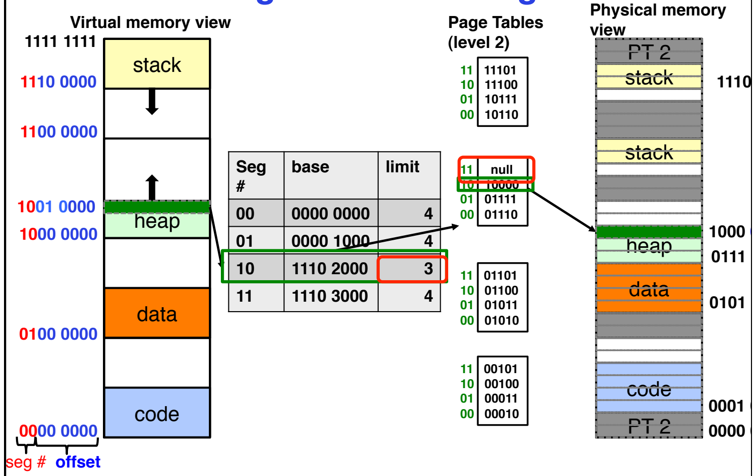
11	00101
10	00100
01	00011
00	00010

Review: Segmentation & Page Tables



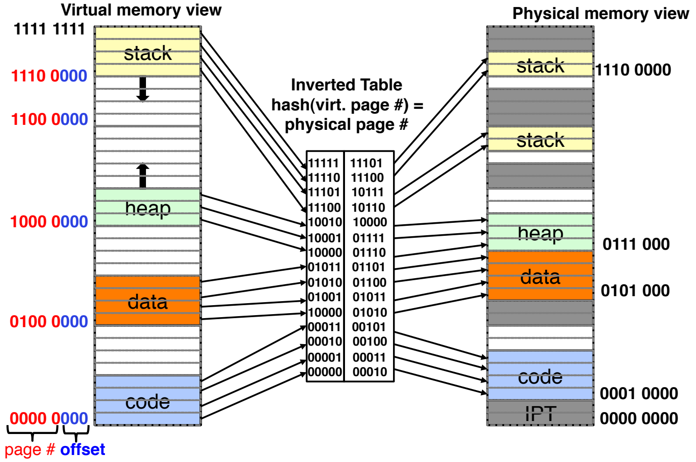
Seg #	base	limit
00	0000 0000	4
01	0000 1000	4
10	1110 2000	3
11	1110 3000	4

Review: Segmentation & Page Tables

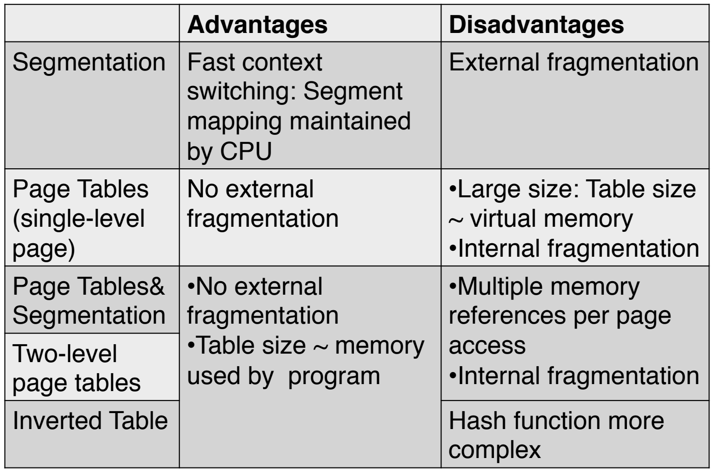


Seg #	base	limit
00	0000 0000	4
01	0000 1000	4
10	1110 2000	3
11	1110 3000	4

Review: Inverted Page Table



Address Translation Comparison



	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Page Tables (single-level page)	No external fragmentation	•Large size: Table size ~ virtual memory •Internal fragmentation
Page Tables& Segmentation	•No external fragmentation •Table size ~ memory used by program	•Multiple memory references per page access •Internal fragmentation
Two-level page tables		
Inverted Table		Hash function more complex

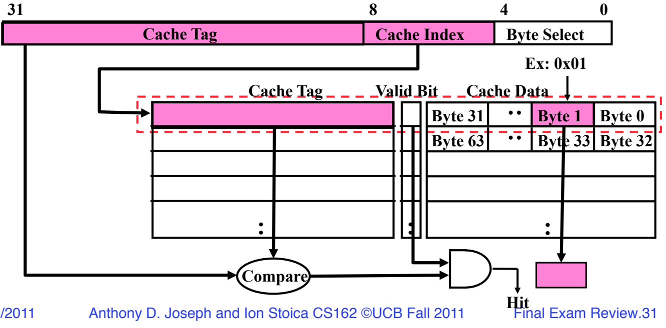
Caches, TLBs

Review: Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - – "Cold" fact of life: not a whole lot you can do about it
  - – Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - – Cache cannot contain all blocks access by the program
  - – Solution: increase cache size
- Conflict (collision):
  - – Multiple memory locations mapped to same cache location
  - – Solutions: increase cache size, or increase associativity
- Two others:
  - – Coherence (Invalidation): other process (e.g., I/O) updates memory
  - – Policy: Due to non-optimal replacement policy

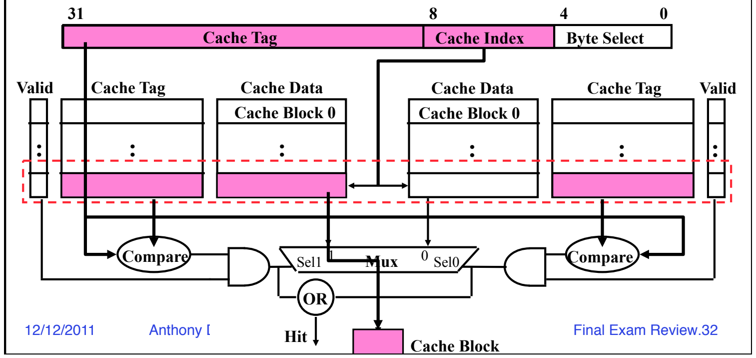
Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - – Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache index" shares the same cache entry
  - – Conflict misses



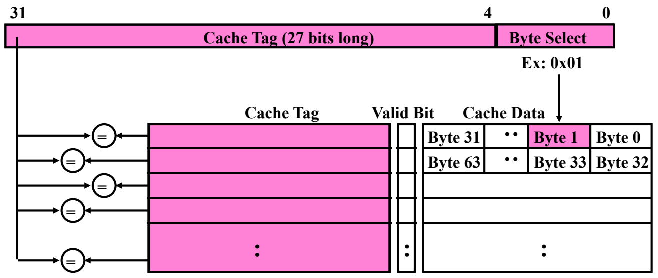
Set Associative Cache

- N-way set associative: N entries per Cache Index
  - – N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - – Two tags in the set are compared to input in parallel
  - – Data is selected based on the tag result



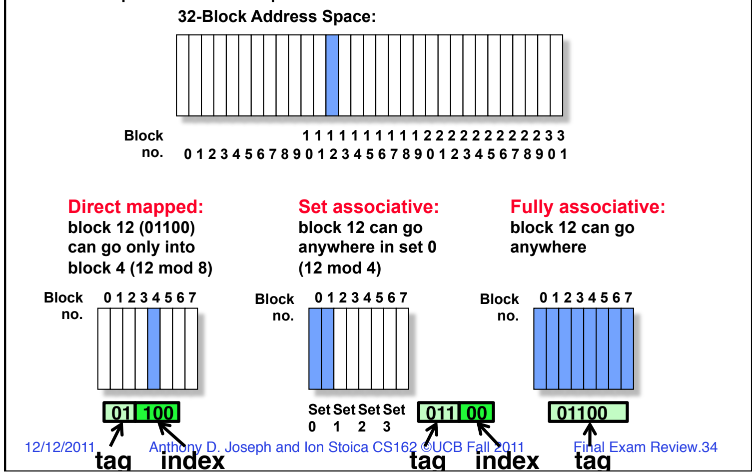
Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



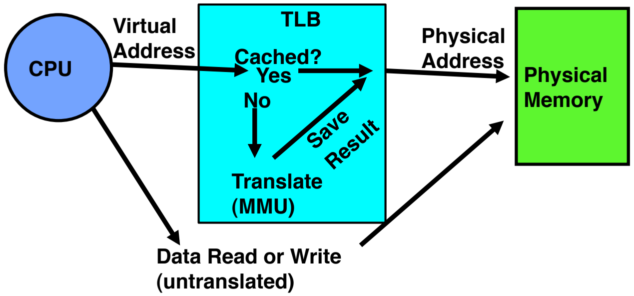
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache



Review: Caching Applied to Address Translation

- Problem: address translation expensive (especially multi-level)
- Solution: cache address translation (TLB)
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…

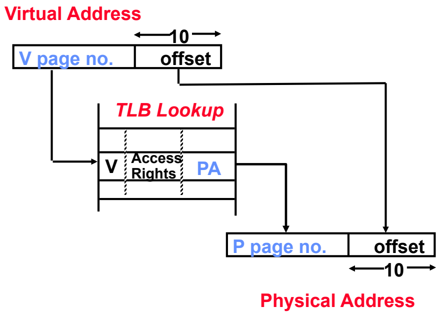


TLB organization

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- When does TLB lookup occur?
  - Before cache lookup?
  - In parallel with cache lookup?

Reducing translation time further

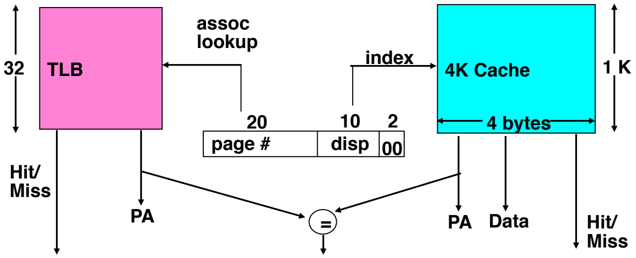
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

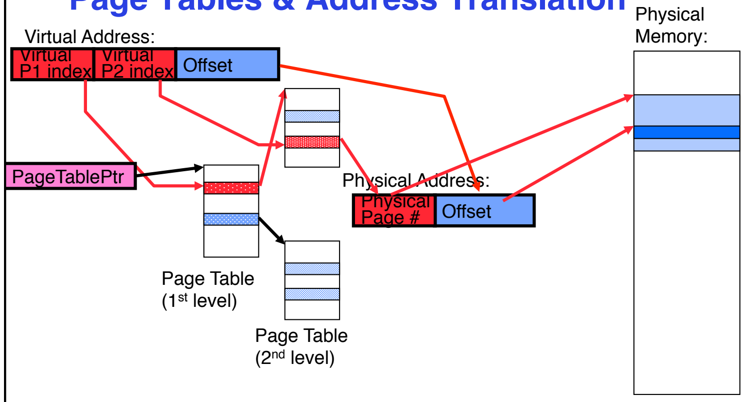
- Here is how this might work with a 4K cache:



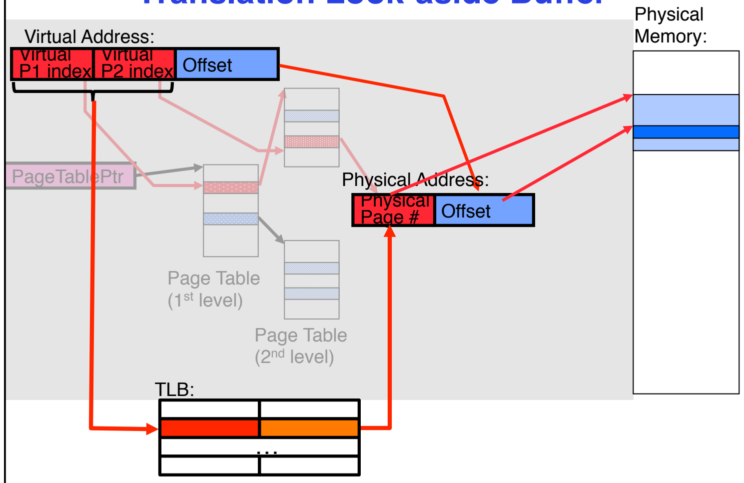
- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

Putting Everything Together

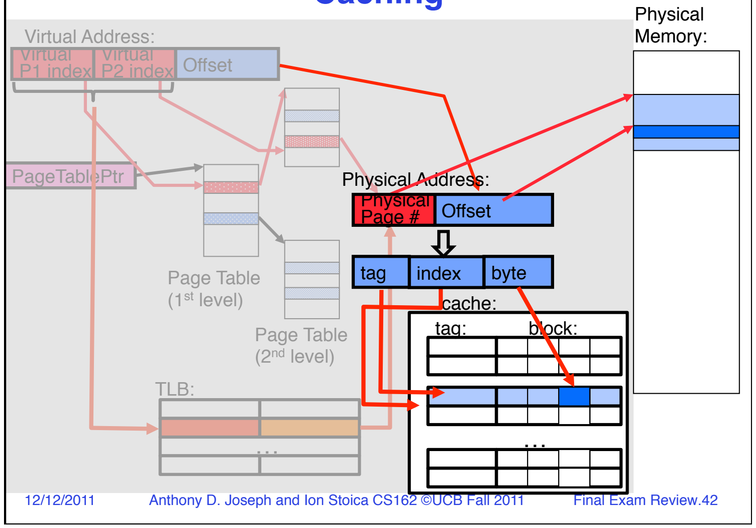
Page Tables & Address Translation



Translation Look-aside Buffer



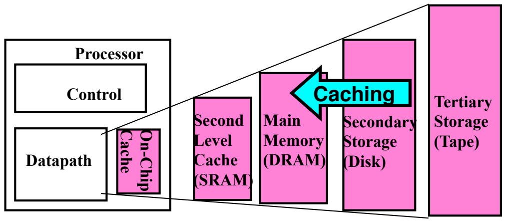
Caching



Demand Paging

Demand Paging

- Modern programs require a lot of physical memory
  - – Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - – 90-10 rule: programs spend 90% of their time in 10% of their code
  - – Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk

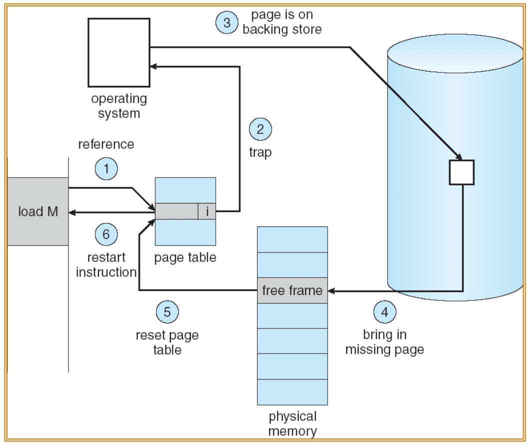


Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Cache

Steps in Handling a Page Fault

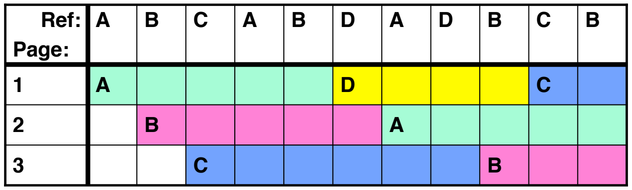


Page Replacement Policies

- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:



Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- MIN Does much better:

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Belady's anomaly)

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Implementing LRU & Second Chance

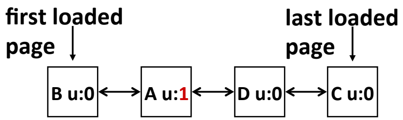
- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Second Chance Algorithm:
  - Approximate LRU
    - » Replace an old page, not the oldest page
  - FIFO with "use" (reference) bit
- Details
  - A "use" bit per physical page
  - On page fault check page at head of queue
    - » If use bit=1 → clear bit, and move page at tail (give the page second chance!)
    - » If use bit=0 → replace page
  - Moving pages to tail still complex

Clock Algorithm

- Clock Algorithm: more efficient implementation of second chance algorithm
  - Arrange physical pages in circle with single clock hand
- Details:
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave it alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
- What if hand moving slowly?
  - Good sign or bad sign?
    - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set

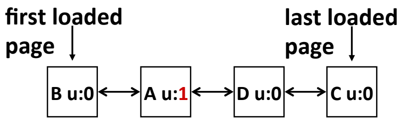
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



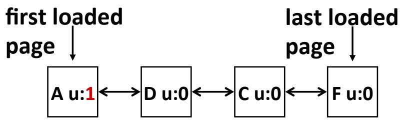
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



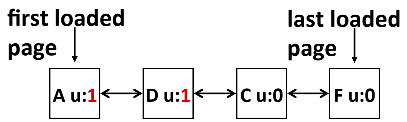
Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



first loaded page

last loaded page

A u:1

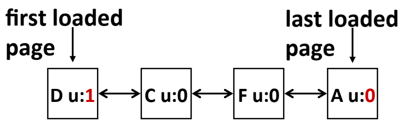
D u:1

C u:0

F u:0

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



first loaded page

last loaded page

D u:1

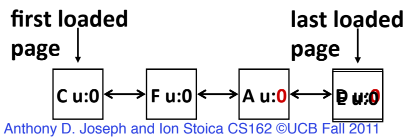
C u:0

F u:0

A u:0

Second Chance Illustration

- Max page table size 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



first loaded page

last loaded page

C u:0

F u:0

A u:0

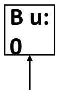
D u:0

E u:0

Clock Replacement Illustration

- Max page table size 4

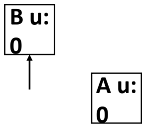
- Invariant: point at oldest page
  - Page B arrives



B u: 0

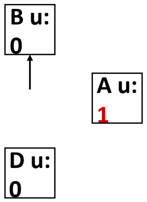
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A



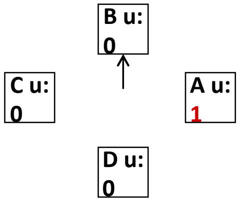
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives



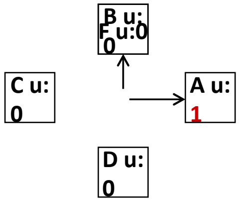
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



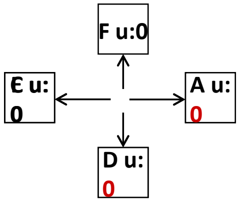
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



Clock Replacement Illustration

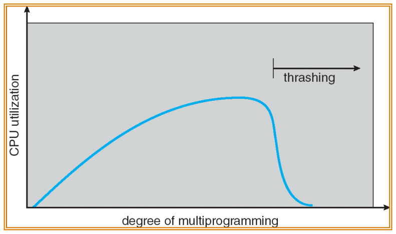
- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

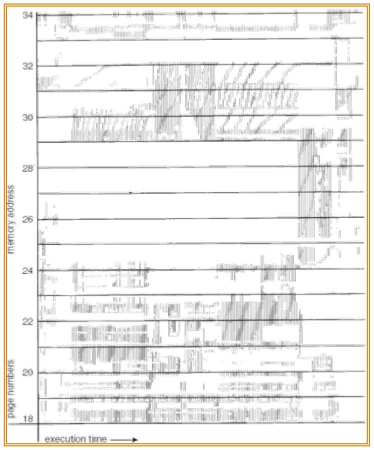
Thrashing



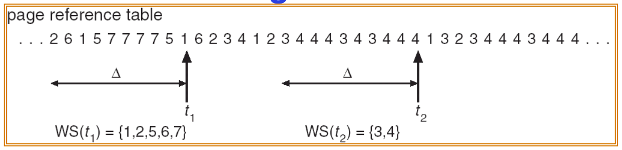
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



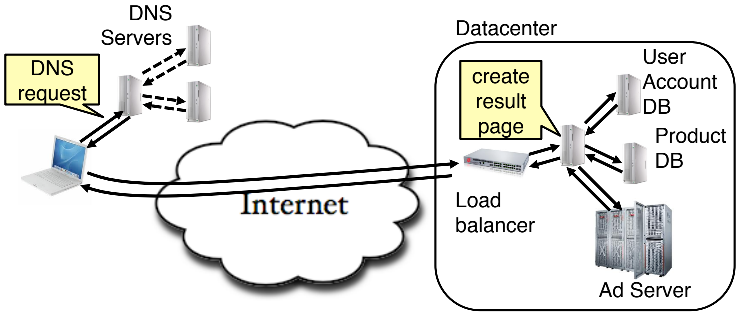
Working-Set Model



- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma|WS_i| \equiv$ total demand frames
- if $D > memory$ ⇒ Thrashing
  - Policy: if $D >$ memory, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

Distributed Systems

Example: Accessing Amazon



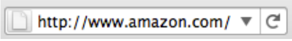
- Complex interaction of multiple components in multiple administrative domains

---

Universal Resource Locator (URL)

`protocol://host-name:port/directory-path/resource`

- This is what you enter in the browser!

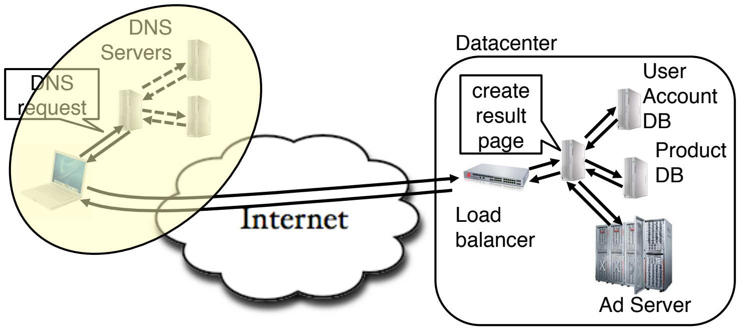


http://www.amazon.com/

- Example:
  - https://www.amazon.com = https://www.amazon.com:443/index.html
  - `protocol` = https
  - `host-name` = www.amazon.com
    - » Name of an Amazon's web server
  - `port` = 443 (default HTTPS port)
    - » Use HTTP over Secure Socket Layer/Transport Layer Security
  - `directory-path` = “”
    - » Path relative to web directory at server (e.g., public_html)
  - `resource` = index.html (default file)
    - » Contains HTML home page of Amazon

Domain Name Service (DNS) Resolution

- Resolve www.amazon.com to the IP address of an Amazon HTTPS server



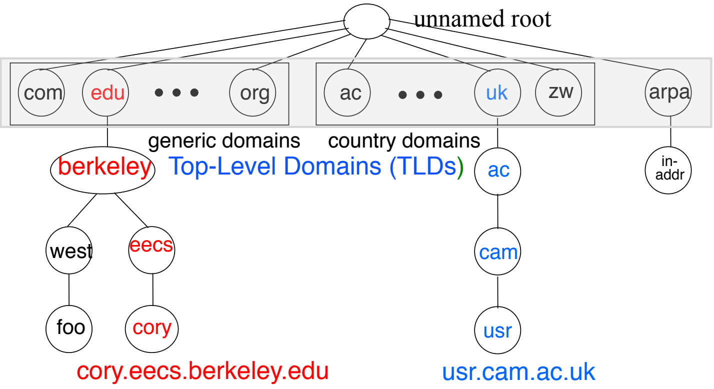
DNS Resolution

- Resolve www.amazon.com to the IP address of an Amazon HTTP server
- How does client know DNS server
  - Client configured with the address of the local DNS server

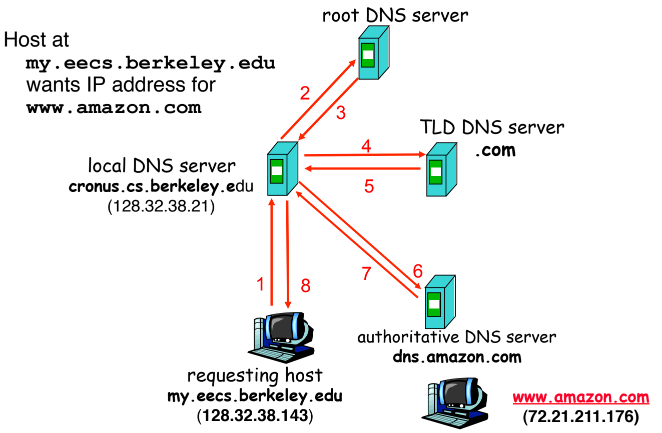
Domain Name System (DNS)

- Properties of DNS
  - Hierarchical name space divided into zones
  - Zones distributed over collection of DNS servers
- Hierarchy of DNS servers
  - Root (hardwired into other servers)
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translations
  - Local DNS servers
  - Resolver software

Distributed Hierarchical Database



Example



Host at
`my.eecs.berkeley.edu`
wants IP address for
`www.amazon.com`

root DNS server

TLD DNS server
.com

local DNS server
cronus.cs.berkeley.edu
(128.32.38.21)

authoritative DNS server
dns.amazon.com

requesting host
my.eecs.berkeley.edu
(128.32.38.143)

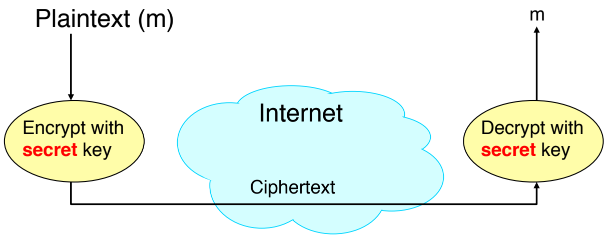
www.amazon.com
(72.21.211.176)

How do You Secure your Credit Card?

- Use a secure protocol, e.g., HTTPS
- Need to ensure three properties:
  - Confidentiality: an adversary cannot snoop the traffic
  - Authentication: make sure you indeed talk with Amazon
  - Integrity: an adversary cannot modify the message
    - Used for improving authentication performance
- Cryptography based solution:
  - General premise: there is a key, possession of which allows decoding, but without which decoding is infeasible
    - Thus, key must be kept secret and not guessable

Symmetric Keys

- Sender and receiver use the same key for encryption and decryption
- Examples: AES128, DES, 3DES



Plaintext (m)

Encrypt with **secret** key

Internet

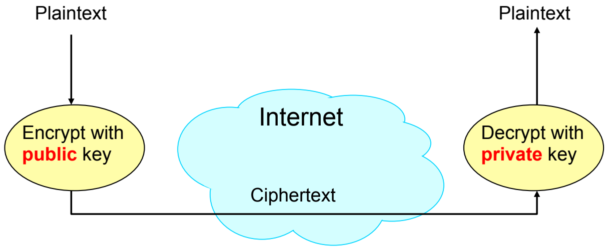
Ciphertext

Decrypt with **secret** key

m

Public Key / Asymmetric Encryption

- Sender uses receiver's public key
  - Advertised to everyone
- Receiver uses complementary private key
  - Must be kept secret
- Example: RSA



Plaintext

Encrypt with **public** key

Internet

Ciphertext

Decrypt with **private** key

Plaintext

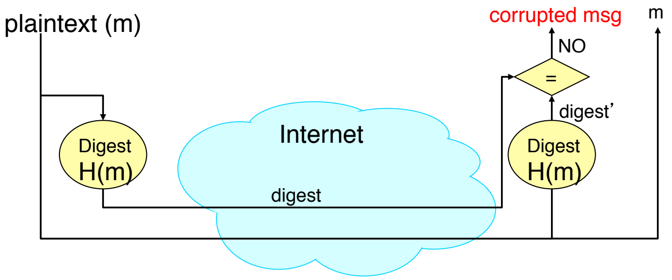
Symmetric vs. Asymmetric Cryptography

- Symmetric cryptography
  - +Low overhead, fast
  - – Need a secret channel to distribute key
- Asymmetric cryptography
  - +No need for secret channel; public key known by everyone
  - +Provable secure
  - – Slow, large keys (e.g., 1024 bytes)

Integrity

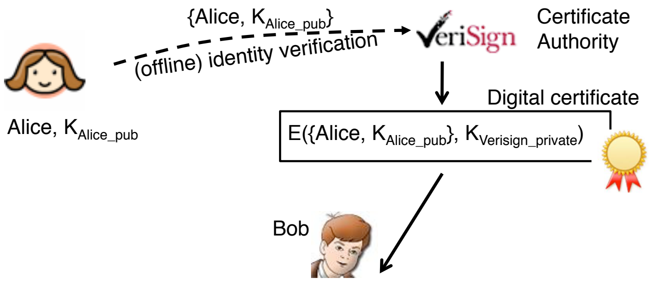
- Basic building block for integrity: *hashing*
  - – Associate hash with byte-stream, receiver verifies match
    - » Assures data <u>hasn't been modified</u>, either accidentally - or maliciously
- Approach:
  - \- Sender computes a *digest* of message m, i.e., H(m)
    - » H() is a publicly known *hash function*
  - \- Send digest (d = H(m)) to receiver in a secure way, e.g.,
    - » Using another physical channel
    - » Using encryption (e.g., Asymmetric Key)
  - \- Upon receiving m and d, receiver re-computes H(m) to see whether result agrees with d
- \- Examples: ~~MD5~~, SHA1

Operation of Hashing for Integrity



Digital Certificates

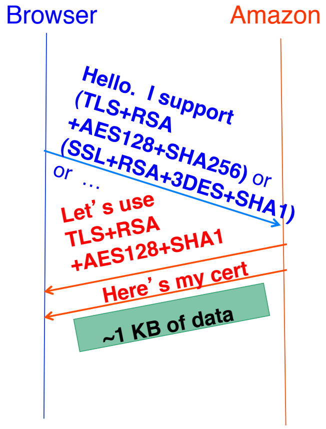
- How do you know **K_{Alice_pub}** is indeed **Alice**'s public key?
- Main idea: trusted authority signing binding between Alice and its private key



D(E({Alice, K_{Alice_pub}}, $K_{Verisign\_private}$), $K_{Verisign\_public}$) = {Alice, K_{Alice_pub}}

HTTPS Connection (SSL/TLS)

- Browser (client) connects via TCP to Amazon's `HTTPS` server
- Client sends over list of crypto protocols it supports
- Server picks protocols to use for this session
- Server sends over its certificate
- (all of this is in the clear)



Browser

Amazon

Hello. I support (TLS+RSA +AES128+SHA256) or (SSL+RSA+3DES+SHA1) or …

Let's use TLS+RSA +AES128+SHA1

Here's my cert

~1 KB of data

Inside the Server's Certificate

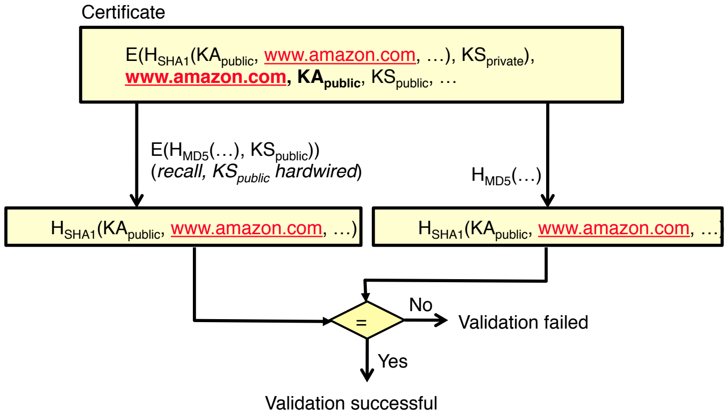
- Name associated with cert (e.g., Amazon)
- Amazon's RSA public key
- A bunch of auxiliary info (physical address, type of cert, expiration time)
- Name of certificate's signatory (who signed it)
- A public-key signature of a hash (SHA1) of all this
  - Constructed using the signatory's private RSA key, i.e.,
  - Cert = $E(H_{SHA1}(KA_{public}$, www.amazon.com, …$), KS_{private})$
    - KA_{public}: Amazon's public key
    - $KS_{private}$: signatory (certificate authority) public key
- …

Validating Amazon's Identity

- How does the browser authenticate certificate signatory?
  - Certificates of few certificate authorities (e.g., Verisign) are hardwired into the browser
- If it can't find the cert, then warns the user that site has not been verified
  - And may ask whether to continue
  - Note, can still proceed, just without authentication
- Browser uses public key in signatory's cert to decrypt signature
  - Compares with its own SHA1 hash of Amazon's cert
- Assuming signature matches, now have high confidence it's indeed Amazon …
  - … assuming signatory is trustworthy

Certificate Validation

- You (browser) want to make sure that KA_{public} is indeed the public key of www.amazon.com



Certificate

$E(H_{SHA1}(KA_{public}$, www.amazon.com, …$), KS_{private})$, **www.amazon.com**, $\mathbf{KA_{public}}$, KS_{public}, …

$E(H_{MD5}(\ldots), KS_{public}))$ (*recall,* KS_{public} *hardwired*)

$H_{MD5}(\ldots)$

$H_{SHA1}(KA_{public}$, www.amazon.com, …$)$

$H_{SHA1}(KA_{public}$, www.amazon.com, …

=

No

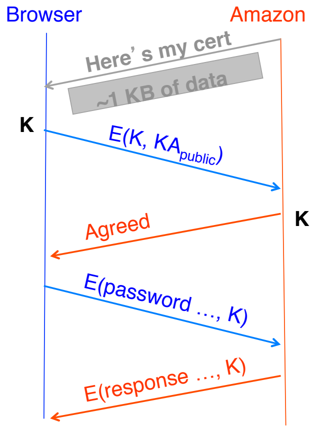
Validation failed

Yes

Validation successful

HTTPS Connection (SSL/TLS), con’t

- Browser constructs a random *session (symmetric) key* K
- Browser encrypts K using Amazon’s public key
- Browser sends E(K, KA_{public}) to server
- Browser displays 🔒
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) using key K
  - E.g., client can authenticate using a password

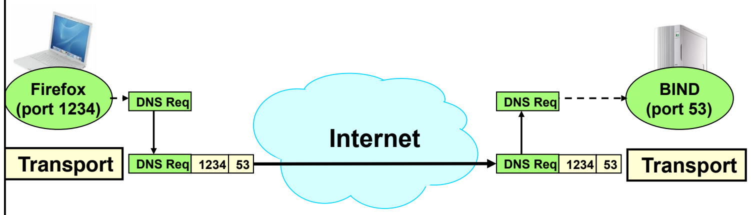


How Does a Client Communicate with Servers?

- A: Via **transport** protocol (e.g., UDP, TCP, …)
- Transport protocol in a nutshell:
  - Allow two application end-points to communicate
    - Each application identified by a port number on the machine it runs
  - Multiplexes/demultiplexes packets from/to different processes using port numbers
  - Can provide reliability, flow control, congestion control
- Two main transport protocols in the Internet
  - **User datagram protocol (UDP):** just provide multiplexing/demultiplexing, no reliability
  - **Transport Control Protocol (TCP):** provide reliability, flow control, congestion control

Transport Layer

- DNS server runs at a specific port number, i.e., 53
  - Most popular DNS server: BIND (Berkeley Internet Name Domain)
  - Assume client (browser) port number 1234

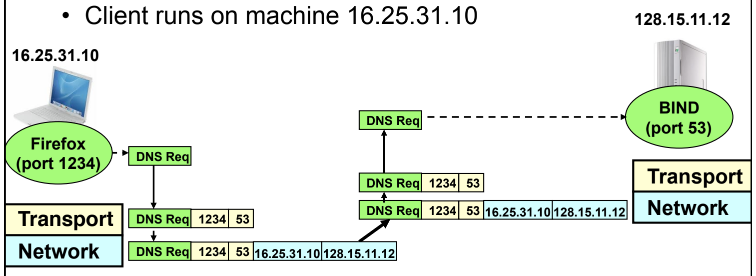


How do UDP packets Get to Destination?

- A: Via network layer, i.e., Internet Protocol (IP)
- Implements datagram packet switching
  - Enable two end-hosts to exchange packets
    - Each end-host is identified by an IP address
    - Each packets contains destination IP address
    - **Independently** routes each packet to its destination
  - **Best effort service**
    - No deliver guarantees
    - No in-order delivery guarantees

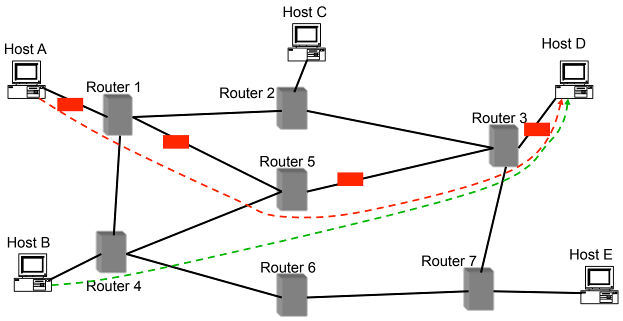
Network (IP) Layer (cont'd)

- Assume DNS server runs on machine 128.15.11.12
  - Client configured with DNS server IP address
- Client runs on machine 16.25.31.10



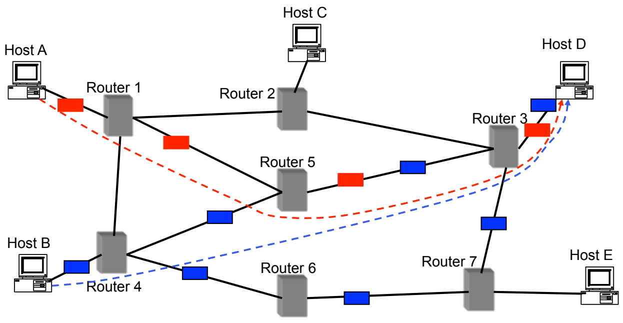
IP Packet Routing

- Each packet is individually routed



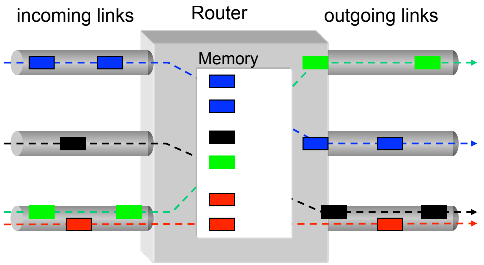
IP Packet Routing

- Each packet is individually routed



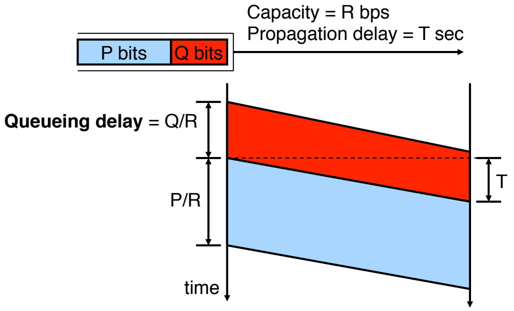
Packet Forwarding

- Packets are first stored before being forwarded
  - Why?

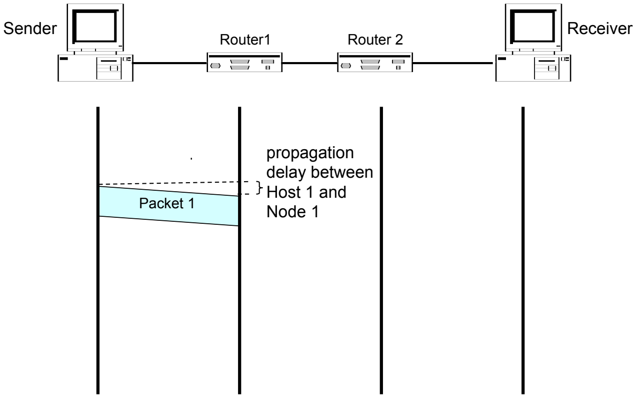


Packet Forwarding Timing

- The queue has Q bits when packet arrives → packet has to wait for the queue to drain before being transmitted

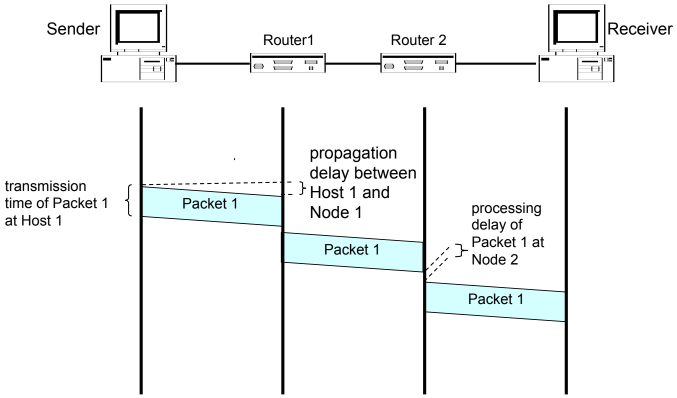


Packet Forwarding Timing



propagation delay between Host 1 and Node 1

Packet Forwarding Timing

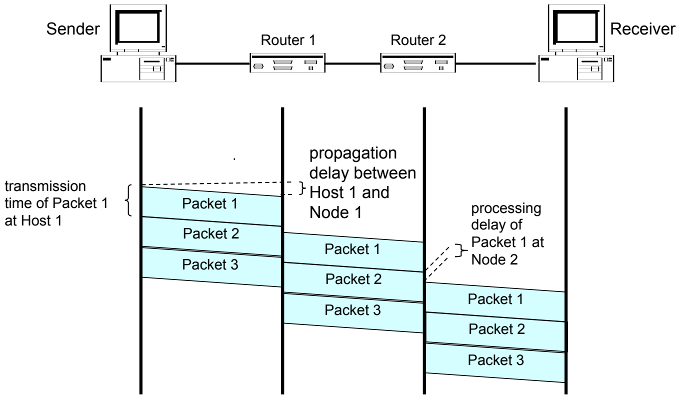


transmission time of Packet 1 at Host 1

propagation delay between Host 1 and Node 1

processing delay of Packet 1 at Node 2

Packet Forwarding Timing

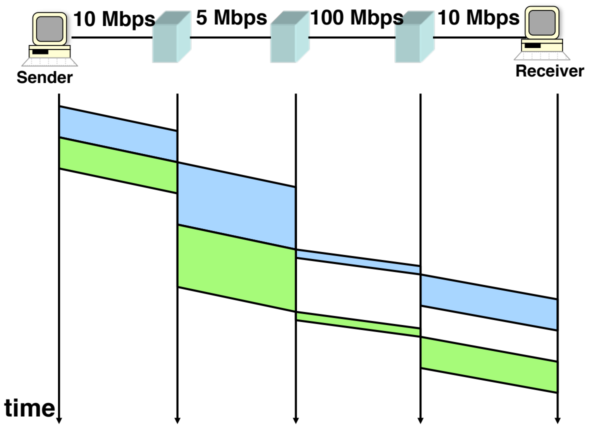


transmission time of Packet 1 at Host 1

propagation delay between Host 1 and Node 1

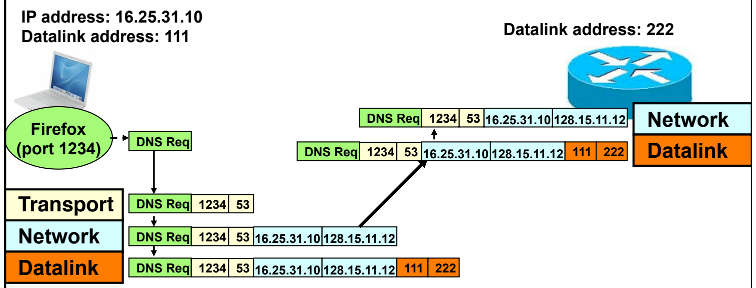
processing delay of Packet 1 at Node 2

Packet Forwarding Timing: Packets of Different Lengths



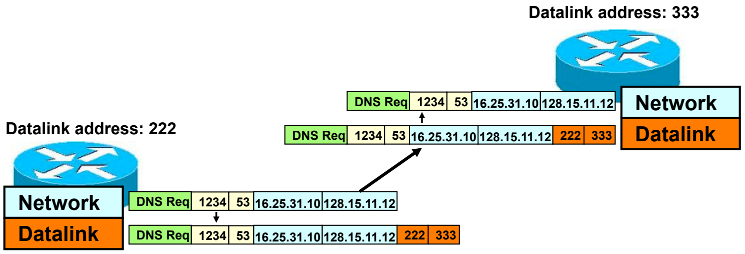
Datalink Layer

- Enable nodes (e.g., hosts, routers) connected by **same link** to exchange packets (frames) with each other
  - Every node/interface has a datalink layer address (e.g., 6 bytes)
  - No need to route packets, as each node on same link receives packets from everyone else on that link (e.g., WiFi, Ethernet)



Datalink Layer

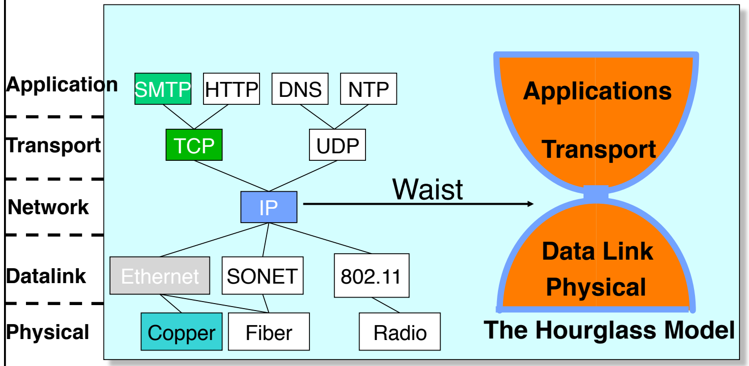
- Enable nodes (e.g., hosts, routers) connected by **same link** to exchange packets (frames) with each other
  - Every node/interface has a datalink layer address (e.g., 6 bytes)
  - **Network layer** picks the next router for the packet towards destination based on its destination IP address



Physical Layer

- Move bits of information between two systems connected by a physical link
- Specifies how bits are represented (encoded), such as voltage level, bit duration, etc
- Examples: coaxial cable, optical fiber links; transmitters, receivers

The Internet *Hourglass*



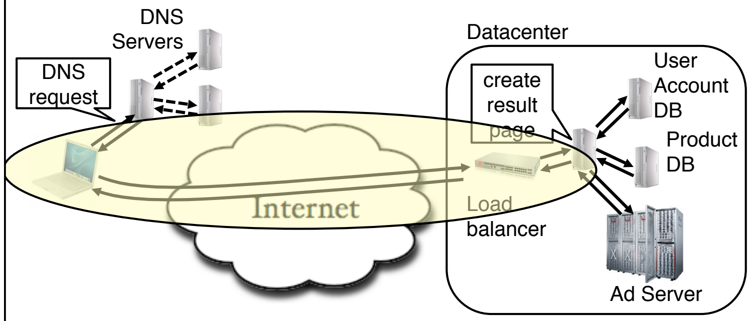
There is just one network-layer protocol, **IP**
The "narrow waist" facilitates interoperability

Implications of Hourglass & Layering

Single Internet-layer module (**IP**):

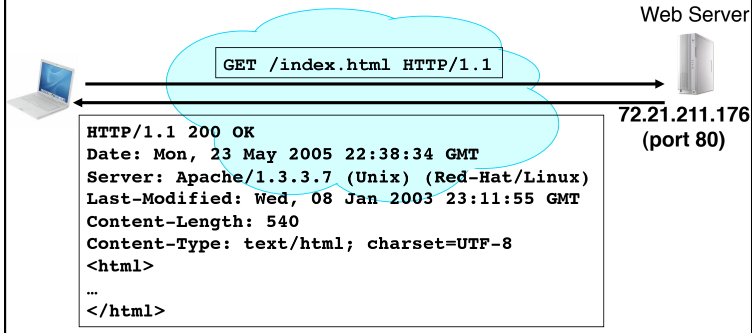
- Allows arbitrary networks to interoperate
  - Any network technology that supports IP can exchange packets
- Allows applications to function on all networks
  - Applications that can run on IP can use any network technology
- Supports simultaneous innovations above and below IP
  - But changing IP itself, i.e., **IPv6**, very involved

HTTP (HyperText Transport Protocol)



HTTP Request

- After resolving DNS request for www.amazon.com to 72.21.211.176 client sends an http GET request to the web server
- Web server returns HTML file for home page

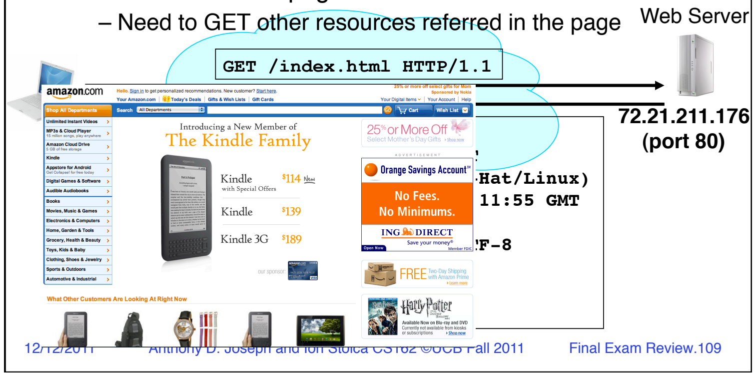


```
GET /index.html HTTP/1.1
```

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Length: 540
Content-Type: text/html; charset=UTF-8
<html>
…
</html>
```

HTTP Request

- After resolving DNS request for www.amazon.com client sends an http GET request to the web server
- Web server returns HTML file for home page
- Client renders the page
  - Need to GET other resources referred in the page

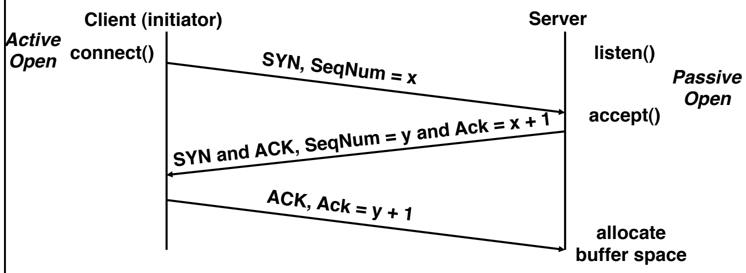


HTTP over TCP

- HTTP runs over TCP not UDP
  - Why?
- TCP: stream oriented protocol
  - Sender sends a stream of bytes, not packets (e.g., no need to tell TCP how much you send)
  - Receiver reads a stream of bytes
- Provides reliability, flow control, congestion control
  - **Flow control:** avoid the sender from overwhelming the receiver
  - **Congestion control:** avoid the sender from overwhelming the network

TCP Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters: the start sequence number for each side
  - Starting sequence numbers are random



TCP Flow Control & Reliability

- Sliding window protocol at byte (not packet) level
  - Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)
- Reliability
  - The ack(nowledgement) contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1
  - Go-back-N: TCP Tahoe, Reno, New Reno
  - Selective acknowledgement: TCP Sack
- We didn't learn about congestion control (two lectures in ee122)

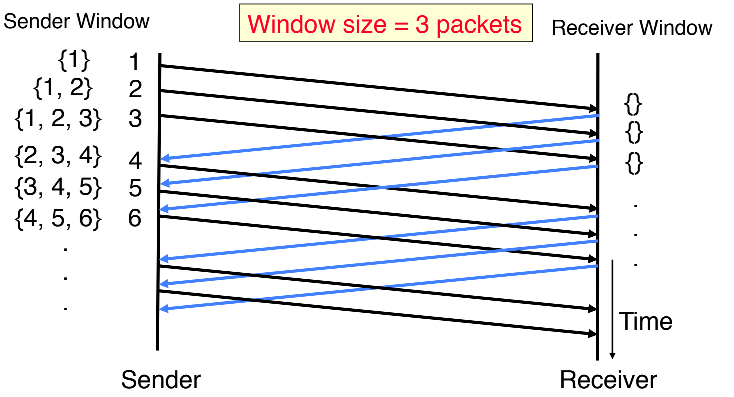
Sliding Window

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*
- Assume window size is n
- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
- Sender can send packets in its window
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
- Receiver can accept out of sequence, if in window

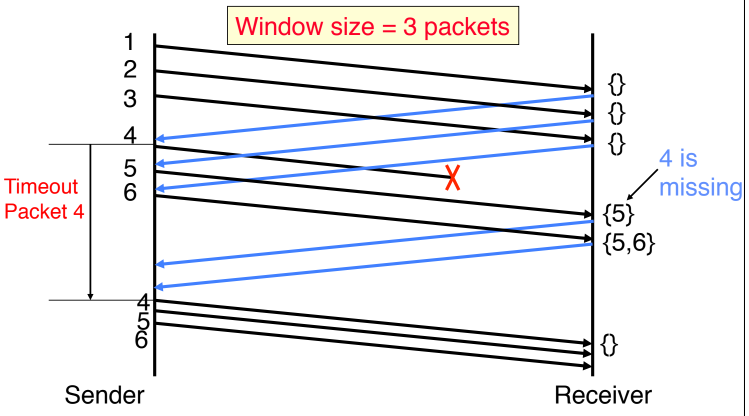
Go-Back-n (GBN)

- Transmit up to *n* unacknowledged packets
- If timeout for ACK(*k*), retransmit *k*, *k+1*, …

GBN Example w/o Errors



GBN Example with Errors



Observations

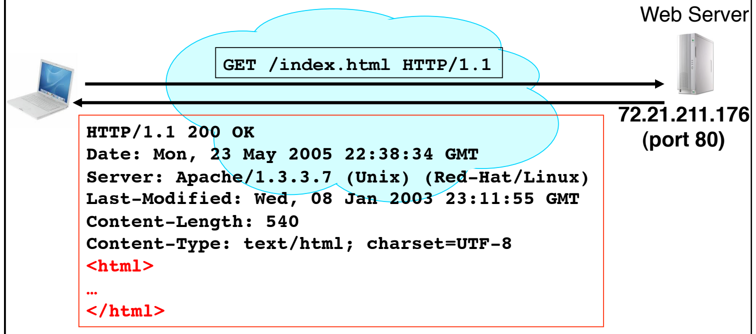
- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough. Throughput is ~ (n/RTT)
  - Stop & Wait is like n = 1.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits

---

Filesystems

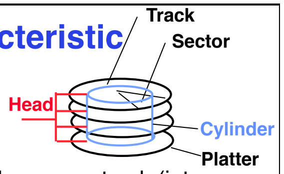
HTTP Server File I/O

- After resolving DNS request for www.amazon.com to 72.21.211.176 client sends an http GET request to the web server
- Web server returns HTML file for home page

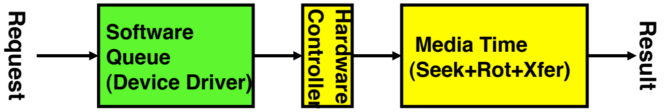


```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Length: 540
Content-Type: text/html; charset=UTF-8
<html>
…
</html>
```

Review: Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queuing Time + Controller time + Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - transfer large group of blocks sequentially from one track

Building a File System

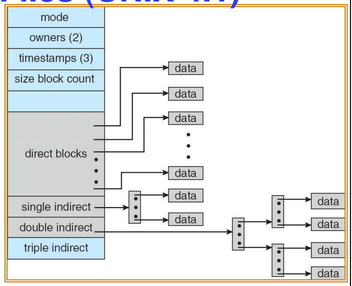
- File System: OS layer that transforms block interface of disks into Files, Directories, etc.

- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability

- How do users access files?
  - Sequential Access: bytes read in order (most file accesses)
  - Random Access: read/write element out of middle of array

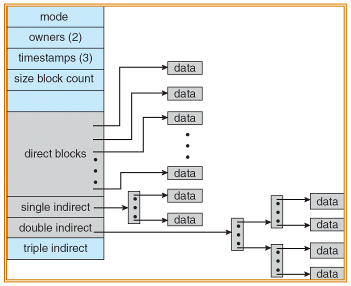
- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)

Multilevel Indexed Files (UNIX 4.1)



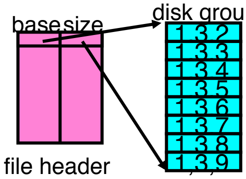
- Multilevel Indexed Files: (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files
- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Example of Multilevel Indexed Files



- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/O's per block!)

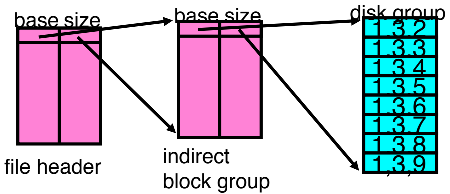
File Allocation for Cray-1 DEMOS



Basic Segmentation Structure:
Each segment contiguous on disk

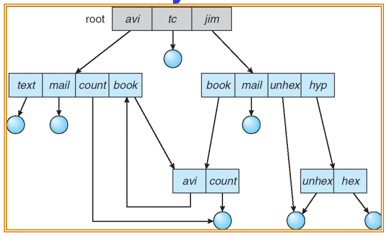
- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed
  - – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - – If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - – Suppose 1000 blocks in a block group ⇒ 80GB max file
    - » Assuming 8KB blocks, 8byte entries⇒ (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - – Pros: Fast sequential access, Free areas merge simply
    Easy to find free block groups (when disk not full)
  - – Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access
  - – Full disk ⇒ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

Directory Structure

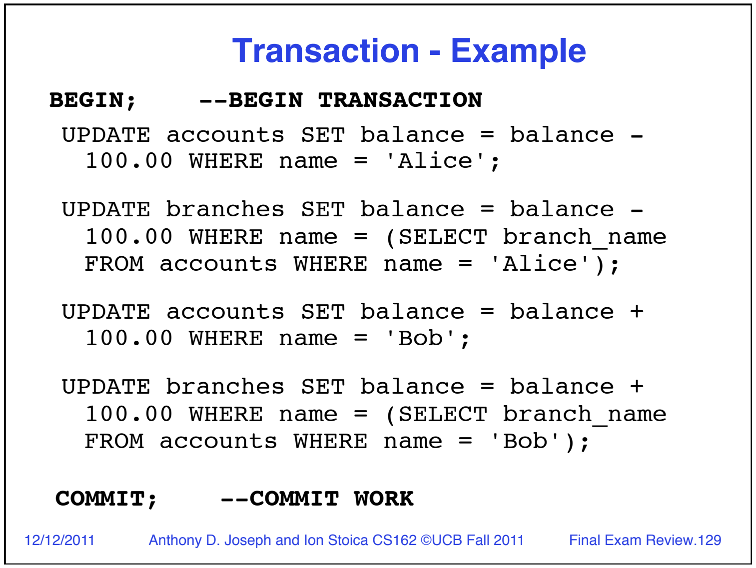


- Not really a hierarchy!
  - – Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - – Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - – Soft Links: "shortcut" pointers to other files
    - » Implemented by storing the logical name of actual file
- Name Resolution: The process of converting a logical name into a physical resource (like a file)
  - – Traverse succession of directories until reach target file
  - – Global file system: May be spread across the network

Transactions

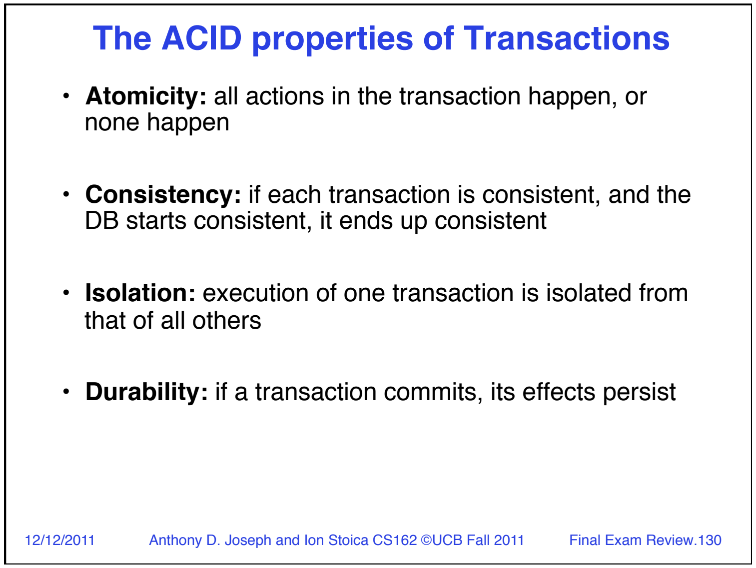
Concurrent Execution & Transactions

- Concurrent execution essential for good performance
  - – Disk slow, so need to keep the CPU busy by working on several user programs concurrently
- DBMS only concerned about what data is read/written from/ to the database
  - – Not concerned about other operations performed by program on data
- **Transaction** – DBMS's abstract view of a user program, i.e., a sequence of reads and writes.



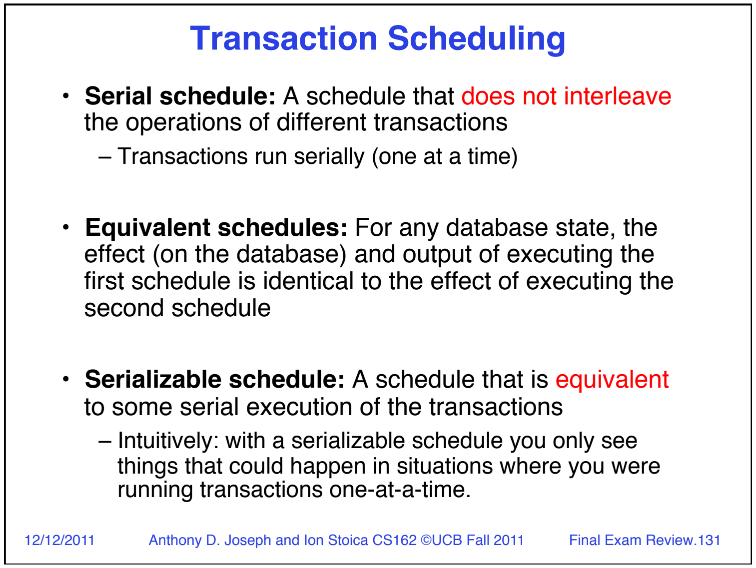
Transaction - Example

```
BEGIN;     --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance -
   100.00 WHERE name = 'Alice';

 UPDATE branches SET balance = balance -
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Alice');

 UPDATE accounts SET balance = balance +
   100.00 WHERE name = 'Bob';

 UPDATE branches SET balance = balance +
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Bob');

COMMIT;     --COMMIT WORK
```



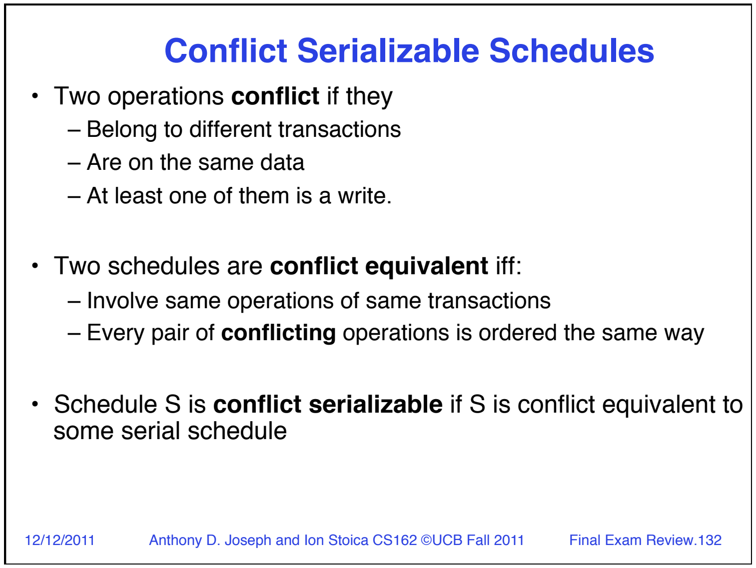
The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** if each transaction is consistent, and the DB starts consistent, it ends up consistent
- **Isolation:** execution of one transaction is isolated from that of all others
- **Durability:** if a transaction commits, its effects persist



Transaction Scheduling

- **Serial schedule:** A schedule that does not interleave the operations of different transactions
  - Transactions run serially (one at a time)
- **Equivalent schedules:** For any database state, the effect (on the database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.



Conflict Serializable Schedules

- Two operations **conflict** if they
  - Belong to different transactions
  - Are on the same data
  - At least one of them is a write.
- Two schedules are **conflict equivalent** iff:
  - Involve same operations of same transactions
  - Every pair of **conflicting** operations is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),         R(B),W(B)
```

⇩

```
T1:R(A),W(A),     R(B),     W(B)
T2:          R(A),     W(A),    R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),    R(B),W(B)
```

Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),    R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),     W(B)
T2:               R(A),     W(A),R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),W(B)
T2:                   R(A), W(A),R(B),W(B)
```

Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**

- Is this schedule serializable?

```
T1:R(A),          W(A)
T2:     R(A),W(A),
```

Dependency Graph

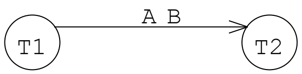
- **Dependency graph:**
  - – Transactions represented as nodes
  - – Edge from Ti to Tj:
    - » an operation of Ti conflicts with an operation of Tj
    - » Ti appears earlier than Tj in the schedule

- **Theorem:** Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

- Conflict serializable schedule:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),         R(B),W(B)
```



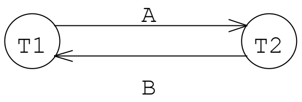
Dependency graph

- No cycle!

Example

- Conflict that is *not* serializable:

```
T1:R(A),W(A),                   R(B),W(B)
T2:          R(A),W(A),R(B),W(B)
```



Dependency graph

- Cycle: The output of T1 depends on T2, and vice-versa

Notes on Conflict Serializability

- Conflict Serializability doesn't allow all schedules that you would consider correct
  - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data
- In practice, Conflict Serializability is what gets used, because it can be done efficiently
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, …
- Two-phase locking (2PL) is how we implement it

Locks

- "Locks" to control access to data
- Two types of locks:
  - shared (S) lock – multiple concurrent transactions allowed to operate on data
  - exclusive (X) lock – only one transaction can operate on data at a time

Lock Compatibility Matrix

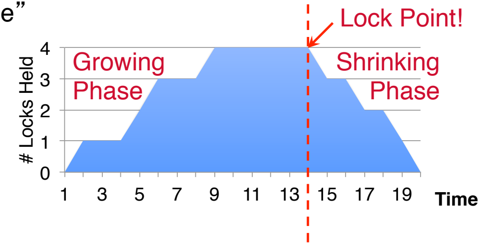
	S	**X**
S	√	–
X	–	–

Two-Phase Locking (2PL)

1) Each transaction must obtain:
  - S (*shared*) or X (exclusive) lock on data before reading,
  - X (*exclusive*) lock on data before writing

2) A transaction can not request additional locks once it releases any locks.

Thus, each transaction has a “growing phase” followed by a “shrinking phase”



Two-Phase Locking (2PL)

- 2PL guarantees conflict serializability

- Doesn’t allow dependency cycles; Why?
- Answer: a cyclic dependency cycle leads to deadlock
  - Edge from Ti to Tj means that Ti acquires lock first and Tj needs to wait
  - Edge from Ti to Tj means that Ti acquires lock first and Tj needs to wait
  - Thus, both T1 and Tj wait for each other → deadlock

- Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by “lock point”

Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits

- If a transaction re-starts, make sure it gets its original timestamp
  - Why?

Example

- T1 transfers $50 from account A to account B

```
T1:Read(A),A:=A-50,Write(A),Read(B),B:=B+50,Write(B)
```

- T2 outputs the total of accounts A and B

```
T2:Read(A),Read(B),PRINT(A+B)
```

- Initially, A = $1000 and B = $2000

- What are the possible output values?

Is this a 2PL Schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	↓
Write(A)	↓
Unlock(A)	↓ <granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
↓	Read(B)
↓ <granted>	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

No, and it is not serializable

Is this a 2PL Schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	↓
Write(A)	↓
Lock_X(B) <granted>	↓
Unlock(A)	↓ <granted>
	Read(A)
	Lock_S(B)
Read(B)	↓
B := B +50	↓
Write(B)	↓
Unlock(B)	↓ <granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Yes, so it is serializable

Cascading Aborts

- Example: T1 aborts
  - Note: this is a 2PL schedule

```
T1:R(A),W(A),           R(B),W(B), Abort
T2:             R(A),W(A)
```

- Rollback of T1 requires rollback of T2, since T2 reads a value written by T1
- Solution: **Strict Two-phase Locking (Strict 2PL)**: same as 2PL except
  - All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont'd)

- All locks held by a transaction are released only when the transaction completes
- In effect, "shrinking phase" is delayed until:
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback).

Is this a Strict 2PL schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	↓ <granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	↓ <granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

No: Cascading Abort Possible

Is this a Strict 2PL schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	↓ <granted>
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)
